Computer Science

Collected Papers of the Soar/IFOR Project Spring 1994

W. Lewis Johnson, Randolph M. Jones, David Keirsey, Frank V. Koss, John E. Laird, Jill F. Lehman, Paul E. Neilsen, Paul S. Rosenbloom, Robert Rubinoff, Karl B. Schwamb, Milind Tambe, Michael van Lent, Robert Wray

April 25, 1994
CMU-CS-94-134

Carnegie Mellon

Collected Papers of the Soar/IFOR Project
Spring 1994

W. Lewis Johnson, Randolph M. Jones, David Keirsey, Frank V. Koss,
John E. Laird, Jill F. Lehman, Paul E. Neilsen, Paul S. Rosenbloom,
Robert Rubinoff, Karl B. Schwamb, Milind Tambe, Michael van Lent, Robert Wray

April 25, 1994
CMU-CS-94-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Since the summer of 1992, the Soar/IFOR research group has been building intelligent automated agents for tactical air simulation. The ultimate goal of this project is to develop automated pilots whose behavior in simulated engagements is indistinguishable from that of human pilots. This technical report is a collection of the research papers that have been generated from this project as of Spring 1994. The research covered in these papers spans a wide spectrum of issues in agent development such as explanation, managing situational awareness, managing multiple interacting goals, coordination between multiple agents, natural language processing, developing believable agents, event tracking, and the infrastructure to support agent development, including knowledge acquisition and use, interfacing to simulation environments, and developing low cost simulators.

This technical report is being published concurrently at the University of Michigan (CSE-TR-207-94), the University of Southern California Information Sciences Institute (ISI/SR-94-379), and Carnegie Mellon University (CMU-CS-94-134).

This research was supported at the University of Michigan, the University of Southern California, and Carnegie Mellon University by contract N00014-92-K-2015 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Research Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, NRL, or the U.S. Government.

Keywords: Artificial intelligence, cognitive simulation, plan execution formation and generation, natural language processing, simulation and modeling applications, simulation support systems

Table of Contents

Preface

Since the summer of 1992, the Soar/IFOR research group has been building intelligent automated agents for tactical air simulation. The Soar/IFOR research project exists at three sites, the University of Michigan, the University of Southern California, and Carnegie Mellon University. The ultimate goal of this project is to develop automated pilots whose behavior in simulated engagements is indistinguishable from that of human pilots. Our work has concentrated on developing agents for beyond visual range engagements where there are either one or two fighter planes on each side.

This technical report is a collection of the research papers that have been generated from this project as of Spring 1994. Most of the papers were presented at the Fourth Conference on Computer Generated Forces and Behavioral Representation in Orlando in May 1994. The others include our paper from the Third Conference on CGF & BR and two papers presented at other workshops and conferences.

The research covered in these papers spans a wide spectrum of issues in agent development such as explanation [3,4], managing situational awareness [5], managing multiple interacting goals [6], coordination between multiple agents [8], natural language processing [9], developing believable agents [11], and event tracking [12]. We have also done research on the infrastructure to support the development of these agents which includes work on knowledge acquisition and use [7], interfacing agents to simulation environments [10], and developing low cost simulators [13]. The papers are organized by having the two overview papers first (the one presented last year followed by the one to be presented this year) [1] & [2], followed by all of the other papers in alphabetic order by author.

1. Jones, R. M., Tambe, M., Laird, J. E., Rosenbloom, P. S. 1993
   Intelligent Automated Agents for Flight Training Simulators.
   Proceedings of the Third Conference on Computer Generated Forces and
   Behavioral Representation. Orlando, FL. pp. 33-42.

2. Rosenbloom, P. S., Johnson, W. L., Jones, R. M., Koss, F., Laird, J. E.,
   Lehman, J. F., Rubinoff, R., Schwamb, K. B., Tambe, M. 1994
   Intelligent Automated Agents for Tactical Air Simulation: A Progress Report.
   Proceedings of the Fourth Conference on Computer Generated Forces and
   Behavioral Representation. Orlando, FL.

3. Johnson, W. L. 1994
   Agents that Explain Their Own Actions.
   Proceedings of the Fourth Conference on Computer Generated Forces and
   Behavioral Representation. Orlando, FL.

4. Johnson, W. L. 1994
   Agents that Learn to Explain Themselves.
   Proceedings of the Twelfth National Conference on Artificial Intelligence, Seattle, WA.

5. Jones, R. M., Laird, J. E. 1994
   Multiple Information Sources and Multiple Participants: Managing
   Situational Awareness in an Autonomous Agent.
   Proceedings of the Fourth Conference on Computer Generated Forces and
   Behavioral Representation. Orlando, FL.

6. Jones, R. M., Laird, J. E., Tambe, M., Rosenbloom, P. S. 1994
Generating Behavior in Response to Interacting Goals.
Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation. Orlando, FL.

7. Koss, F. V., Lehman, J. F. 1994
Knowledge Acquisition and Knowledge Use in a Distributed IFOR Project.
Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation. Orlando, FL.

8. Laird, J. E., Jones, R. M., Nielsen, P. E. 1994
Coordinated Behavior of Computer Generated Forces in TacAir-Soar.
Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation. Orlando, FL.

9. Rubinoff, R., Lehman, J. F. 1994
Natural Language Processing in an IFOR Pilot.
Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation. Orlando, FL.

10. Schwamb, K. B., Koss, F. V., Keirsey, D. 1994
Working with ModSAF: Interfaces for Programs and Users.
Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation. Orlando, FL.

11. Tambe, M., Jones, R. M., Laird, J. E., Rosenbloom, P. S., Schwamb, K. 1994
Building Believable Agents for Simulation Environments: Extended Abstract.
Proceedings of the AAAI Spring Symposium on Believable Agents, 1994.

12. Tambe, M., Rosenbloom, P. S. 1994
Event Tracking in Complex Multi-agent Environments.
Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation. Orlando, FL.

13. van Lent, M., Wray, R. 1994
A Very Low Cost System for Direct Human Control of Simulated Vehicles.
Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation. Orlando, FL.

This technical report is being published concurrently at the University of Michigan (CSE-TR-207-94), the University of Southern California Information Sciences Institute (ISI/SR-94-379), and Carnegie Mellon University (CMU-CS-94-134).

This research was supported at the University of Michigan, the University of Southern California, and Carnegie Mellon University by contract N00014-92-K-2015 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Research Laboratory.

Intelligent Automated Agents for Flight Training Simulators

Randolph M. Jones,[1] Milind Tambe,[2] John E. Laird,[1] and Paul S. Rosenbloom[3]

[1]Artificial Intelligence Laboratory
University of Michigan
1101 Beal Avenue
Ann Arbor, MI 48109-2110

[2]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[3]Department of Computer Science and
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina Del Rey, CA 90292

Abstract

Training in flight simulators will be more effective if the agents involved in the simulation behave realistically. Accomplishing this requires that the automated agents be under autonomous, intelligent control. We are using the Soar cognitive architecture to implement intelligent agents that behave as much like humans as possible. In order to approximate human behavior, the agents must integrate planning and reaction in real time, adapt to new and unexpected situations, learn with experience, and exhibit the cognitive limitations and strengths of humans. This paper describes two simple tactical flight scenarios and the knowledge required for an agent to complete them. In addition, the paper describes an implemented agent model that performs in limited tactical scenarios on three different flight simulators.

The goal of this research is to construct intelligent, automated agents for flight simulators that are used to train navy pilots in flight tactics. When pilots train in tactical simulations, they learn to react to (and reason about) the behaviors of the other agents (friendly and enemy forces) in the training scenario. Thus, it is important that these agents behave as realistically as possible. Standard automated and semi-automated agents can provide this to a limited extent, but trainees can quickly recognize automated agents and take advantages of known weaknesses in their behavior. To provide a more realistic training situation, automated agents should be indistinguishable from other human pilots taking part in the simulation.

To construct such intelligent, automated agents, we have applied techniques from the fields of artificial intelligence and cognitive science. The agents are implemented within the Soar system, a state-of-the-art, integrated cognitive architecture (Rosenbloom et al., 1991). These agents incorporate knowledge gleaned from interviews with experts in flight tactics and analysis of the tactical domain. Soar is a promising candidate for developing agents that behave like humans. Flexible and adaptive behavior is one of Soar's primary strengths, and Soar's learning mechanism provides it with the capability of improving its performance with experience. In addition, Soar allows the smooth integration of planning and reaction in decision making (Pearson et al, 1993). Finally, Soar is the foundation for the development of a proposed unified theory of human cognition (Newell, 1990), and thus maps quite well onto a number of the cognitive issues of interest. This paper reports the results of our research in constructing an intelligent agent for an initial, simple training scenario and our efforts at supplementing the agent's knowledge in order to carry out more complex missions.

Complexities of tactical decision-making

In order to complete a tactical mission, pilots incorporate multiple types of knowledge. These include, for example, knowledge about the goals of the mission, airplane and weapon constraints, survival tactics, controlling the vehicle, characteristics of the environment, and the physical and cognitive capabilities of all of the agents taking part in the scenario. In addition, pilots use their knowledge flexibly and exhibit adaptive behavior. This includes a variety of capabilities, such as reasoning about (and surviving in) unexpected situations, adapting to new situations, learning from experience, and addressing multiple goals simultaneously (e.g., protecting a position, intercepting the enemy, and surviving). Finally, pilots integrate decision-making during a mission with split-second reactions to new situations and potential threats.

Robust automated forces that can carry out general simulated missions must address these issues, especially if the forces are to behave as humans would in similar circumstances. In addition to providing the wide range of capabilities that human pilots exhibit, intelligent agents must reflect the same types of weaknesses as humans. These include mental limitations, such as attention and cognitive load, and physical limitations, such as reduced cognitive processing under high forces (such as during a hard turn).

To capture the complex interactions between agents in a simulation, we feel it necessary for each agent to be as autonomous and intelligent as possible. Simulation via stochastic methods can capture general behaviors of groups of agents, but a more realistic simulation requires each agent to behave individually, with is own set of goals, constraints, and perceptions. In addition, if the agents are to be used for training pilots, they must be intelligent in order to provide as rich a training environment as would flying against real humans.

Requirements for an intelligent automated agent

The primary research question is how intelligent, automated agents should be implemented. A simple solution would be to attempt to create "simulation-pilot expert systems". This would involve converting knowledge about high-level tactical decision-making into a fixed rule base. The system would suggest the most appropriate action (or set of actions) based on the current status of the environment. In fact, a number of expert systems have been implemented for various aspects of tactical decision-making (e.g., Kornell, 1987; Ritter & Feurzeig, 1987; Zytkow & Erickson, 1987;).

However, while expert systems have some of the strengths required for realistic simulation, they are usually weak in other areas. In a standard rule-based approach, it is difficult to capture the complexity of the multiple, dynamic goals that pilots must reason about. In contrast, systems that *can* reason well in such a complex domain generally have difficulties making decisions in real time, and they often do not have the ability to react to changes in the environment when there is not enough time to plan ahead. In addition, systems with only high-level tactical knowledge prove to be rather rigid. Unless the system can be preprogrammed for every possible contingency, its performance degrades greatly when it finds itself in unexpected situations. Finally, expert systems generally ignore the possibility of learning with experience and other cognitive aspects of the task. Intelligent, autonomous agents must combine all of these strengths, having the ability to reason about multiple goals in a complex environment, react quickly and appropriately when the time for complex reasoning is limited, adapt to new situations gracefully, and improve its behavior with experience.

In order to create an agent that can reason and react in real time, and is flexible enough to adapt to new situations, it is not enough simply to encode high-level tactics as rules in the system. Rather, the system must also *understand* why each high-level tactical decision is made, so it must contain knowledge of the first principles that support those decisions. For example, part of one tactic for intercepting a bogey involves achieving a desired lateral separation from the bogey's flight path. One way to generate this behavior is to include a specific rule for the agent to move to the desired lateral separation when it is on the appropriate leg of the intercept. However, a more intelligent agent encodes the knowledge that explains why this particular tactic works (so that the fighter will have enough space to come around for a rear-quarter shot if the long and medium-range missiles miss).

With the appropriate supporting knowledge, the system can function in situations that the programmer may not have anticipated. Maintaining lateral separation from the bogey's flight path is a general principle that allows the fighter room to negotiate a turn for a short-range missile shot. This principle may have an impact in a large number of tactical situations, and therefore shouldn't be considered as merely an instruction to follow for one particular type of intercept. If the system reasons from first principles, the programmer does not have to hard code every possible contingency, and good variations on tactics should emerge in response to unanticipated changes in the simulation environment.

Implementing the agent in this manner also provides advantages in terms of adding new knowledge to the system. If the tactical decisions emerge from low-level knowledge, high-level decisions will change appropriately as the supporting knowledge is changed or supplemented. New low-level knowledge (such as a better understanding of geometric principles or radar limitations) will interact with existing knowledge to generate subtle (or possibly dramatic) changes in behavior. Thus, the agent can reason in a number of new situations without requiring a new specific rule for each case. The ease of adding new knowledge to the system also makes it possible to incorporate existing machine-learning mechanisms. These can allow the system to adapt and improve its behavior with experience, as well as provide insights into how human pilots learn about tactics.

The Soar architecture for problem solving (Newell, 1990) is well suited for this type of task. It divides knowledge into *problem spaces* and allows goals and actions in one problem space to be implemented via reasoning in another. Thus, when the agent has a high level goal to intercept a bogey, for example, it can switch problem spaces and reason about the characteristics of its weapons, radar, airplane, and military doctrine. The knowledge from each of these spaces combines to generate an appropriate tactical action. In turn, the high-level action can then be implemented in a problem space that contains medium-level knowledge about plane maneuvers or low-level knowledge about moving the stick and flipping switches.

Because knowledge is separated into problem spaces, it can be easily updated. For example, if the agent's plane is equipped with a new radar with a longer range, only the knowledge in the "radar" space need be updated. New decisions made in the radar space will interact with the results of reasoning in other problem spaces, eventually impacting high-level decisions such as which specific actions should be taken to intercept a bogey. Likewise, if the automated agent is moved to a new simulation environment with a new interface, we can appropriately update the knowledge in the "control" problem space, leaving the remaining knowledge intact.

Simple tactical situations

Our initial effort to construct an intelligent agent focuses on two tactical scenarios used in training pilots: the "non-jinking bogey" and "1-v-1 aggressive bogey" scenarios. In the non-jinking bogey scenario, the target is an airplane (such as a cargo or fuel plane) that holds a steady course and altitude, and does not carry any offensive threats. The key to this scenario is that the bogey does not attempt to evade (jink) the fighter's attack in any way. Although this situation is not likely to occur often in real combat situations, it is a valuable training situation for pilots. It teaches them how to line up the delivery of various types of missiles when the bogey's behavior is very predictable. When a non-offensive bogey's behavior becomes less predictable, the tactics required to intercept it actually become simpler (but less effective).

There are three main phases involved in attacking a non-jinking bogey (see Figure 1). These involve delivering long, medium, and short-range missiles. During each of the phases, the fighter must assume that the current missile will miss, and simultaneously maneuver into the most advantageous position for the next phase. For example, while moving closer to the bogey to fire a long-range missile, the fighter also attempts to achieve the best lateral separation and target aspect for a shot with the medium-range missile (see Figure 2). After delivering a medium-range missile, the fighter must perform displacement and counter turns in order to end up behind the bogey. This allows the fighter to fire a rear-quarter short-range missile. Due to these constraints, the fighter cannot simply head on a collision course with the bogey, but must get to the bogey as quickly as possible while ensuring that it can eventually achieve a rear-quarter missile shot.

The tactics for executing this scenario are relatively simple. The fighter must achieve the appropriate lateral separation and target aspect while firing its weapons at the right times. Then it must execute the dis-

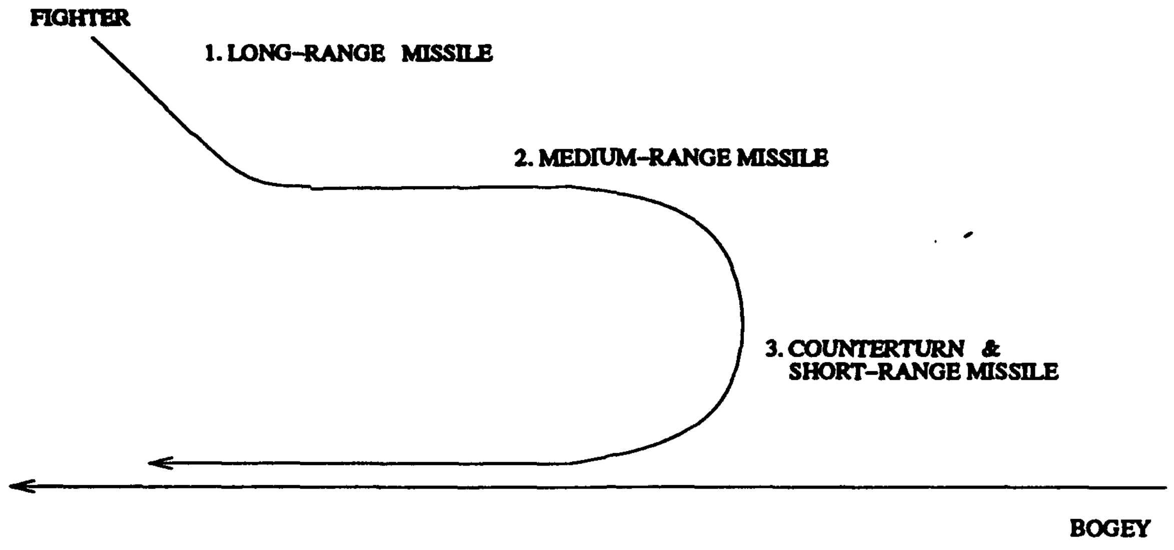


Figure 1. Three stages for intercepting a non-jinking bogey.

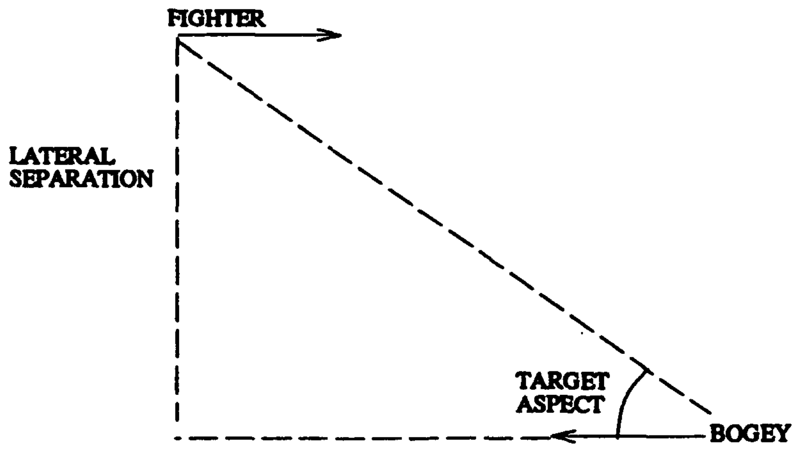


Figure 2. Definition of lateral separation and target aspect.

placement and counter turns and deliver the short-range missile. As mentioned previously, we could code these tactics directly into rules for the agent, but they would then only work under very specific circumstances where everything goes right. Thus, we have implemented the knowledge that supports these tactics. This knowledge justifies *why* each tactical decision should be made when it is made. This allows the system, for example, to get back on course for a short-range missile shot if it misses its opportunity for the medium-range missile shot for some reason. In addition, any particular action that the agent generates will be based on the supporting knowledge, and the agent has the potential to explain its decision (a facility we plan to add in the future).

The 1-v-1 aggressive bogey scenario involves two airplanes with similar capabilities. One is protecting a high-value unit and the other is attempting to destroy it. When the two fighters come in contact they both attempt to intercept and destroy each other, with the overall goal of surviving. This scenario highlights an interaction between different low-level constraints that results in tactical decisions. For example, if one fighter is equipped with a slightly better radar, missiles with longer range, or a more mobile airplane, it dramatically affects the actions that should be taken in completing the mission and surviving. Our agent so far only partially implements this 1-v-1 scenario, and it involves a number of issues that make it more complex than the non-jinking bogey scenario. After discussing the current state of the agent model, we will describe these issues in detail.

Details of the intelligent agent

In order to construct an agent that successfully intercepts a non-jinking bogey, we analyzed tactics for the scenario and interviewed former pilots and radar intercept officers. This allowed us to determine the underlying knowledge and first principles that support the tactics. Then, we encoded this knowledge into an executable Soar system.

The Soar agent's knowledge is organized into problem spaces, each containing operators that allow the agent to reason about particular types of goals. When the agent cannot immediately carry out an action at one level, it uses Soar's universal subgoaling mechanism to move into an alternate problem space and consider methods for carrying out that action. Therefore, high-level tactical decisions are eventually implemented as medium-level maneuver actions or low-level control actions, and the agent always has multiple goals in memory that it uses to reason about and react to its ever-changing situation.

Depending on the particular simulation platform, the current Soar agent requires between 13 and 17 problem spaces to reason with; i.e., 13–17 different types of goals that it reasons about. Most of these are shown in Figure 3. The mission, protect-hvu, barcap, and intercept problem spaces encode tactical knowledge for carrying out missions and performing intercepts. The problem spaces for weapons and missiles include knowledge about specific weapons and the actions that must be performed to deliver them to a target. The maneuver and absolutes problem spaces determine the actual plane maneuvers that must be carried out to implement higher-level actions. The remaining problem spaces im-

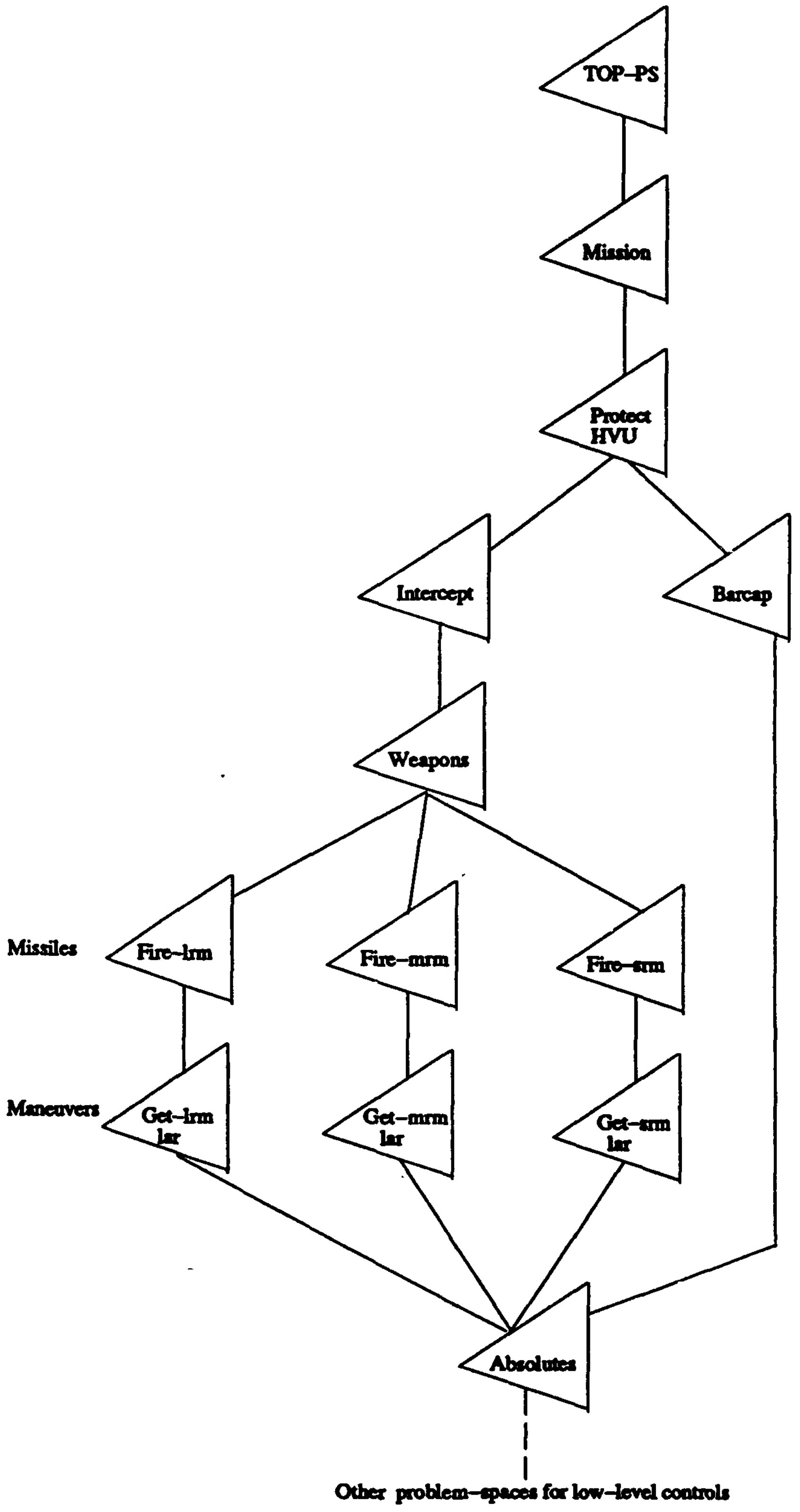


Figure 3. Soar's problem spaces for intercepting a non-jinking bogey.

plement airplane maneuvers at various levels of specification, down to the level of stick and button commands that are issued to the flight simulator.

At any particular instant, between 5 and 12 problem spaces (or hierarchical goals) are usually active. Thus, when changes occur in the agent's situation, there are multiple levels at which the agent may react (Pearson et al., 1993). For example, at a low level, a sudden down draft can cause a change in climb-rate or altitude, leading the agent directly to pull back on the stick. At a higher level, a maneuver by a bogey on the radar can cause a change in tactics. Any reasoning involved in implementing the new tactical decision also percolates down to a new maneuver or stick action. In this manner, Soar maintains its variety of goals in parallel, and violations of the goals at any level lead to immediate action at the appropriate level.

We have implemented an initial model for the non-jinking bogey scenario in whole or in part on three separate flight simulators. The simplest simulator moves planes in a two-dimensional grid-world. In addition, the planes do not move with realistic flight dynamics. We used this simulator to prototype the system and debug the high-level tactics embedded in the system. The second flight simulator was adapted from the flight simulator provided with SGI graphics workstations. It works in real time and requires the agent to issue very low level commands at the level of moving the stick (by issuing mouse pixel movements) and other low-level commands (by simulating keyboard presses). The non-jinking bogey scenario has not yet been completely implemented on this simulator, because Soar must handle the low level intricacies of simply flying the airplane as well as worrying about tactical decisions and maneuvering. Finally, we have implemented the scenario on BBN's ModSAF simulator, which has the most realistic flight dynamics of the three simulators. This simulator works in real time (with a scheduler dividing time between the simulation and agents) and it takes commands at the level of maneuver actions (such as desired heading and altitude) without making the agent concern itself with how the maneuvers are actually implemented with airplane controls.

As of now, we have not completely developed the knowledge base that would allow our agent to successfully fly the 1-v-1 aggressive bogey scenario. This scenario differs from the non-jinking bogey scenario along two major dimensions. First, the bogey maneuvers, so its behavior is not entirely predictable. Second, the bogey is aggressive and has offensive capabilities, so any action that is taken must also address the overall goal of surviving: the agent cannot simply close in on the bogey and shoot it.

In order to successfully complete a mission against an aggressive bogey, the agent must include not only extra knowledge in its tactical problem spaces, but it must also have two new capabilities to address the above issues. First, the agent must be able to interpret and assess its current situation at all (or at least most) times. This primarily involves interpreting the bogey's current actions and predicting its future actions. As with most of the agent's reasoning, the interpretation process also takes place at multiple levels. At a low level, the fighter must recognize when the bogey has initiated a turn and when it has completed one. At a higher level, the fighter must determine whether the turn indicates some kind of threat, and what that threat may be. For example, if the bogey initially comes to a collision course with the fighter, this probably indicates that the bogey is aggressive and is going to try to shoot the fighter. If the bogey points towards the fighter and then makes a hard turn, this indicates that the bogey has probably just fired a missile. The agent must interpret the limited information it gets from its sensors. Then it must use this interpretation to predict the goals that the bogey is trying to achieve and the actions at different levels that the bogey is carrying out.

The second necessary capability for the agent is to use multiple high-level goals to constrain the actions that the agent generates. These types of goals are a bit different from the parallel goals that the Soar agent already handles, because they are not hierarchical in nature. Rather, they are distinct goals that interact with each other. For example one goal, *destroy bogey*, implies that the fighter should close in on the bogey as quickly as possible. However, another goal, *survive*, pressures the fighter to avoid the bogey in order to stay out of the bogey's weapon range. These conflicting goals both must be used to select from multiple possible actions. This type of reasoning leads directly to composite tactical actions. For example, the fighter may get close enough to fire a missile and then make a sudden hard turn. The turn must be hard enough to keep the bogey and fighter from getting close too quickly, but not so hard that the fighter loses its radar lock on the bogey (which would put the fighter at a large disadvantage). In this manner, the agent determines the best action that supports two simultaneous, conflicting goals.

The issues of interpretation and simultaneous goals are not trivial, and they play central roles in agent reasoning for any tactical situations except the simplest ones. Much of tactical decision making involves creating a model of the world from limited information and addressing multiple goals and constraints, such as the current mission, survival, and the characteristics and status of the weapons and airplane. We have not completed the incorporation of this knowledge into the agent yet, but we are taking advantage of the strengths of the Soar architecture in order to implement these two important capabilities (Covrigaru, 1992).

Discussion

We have implemented an intelligent, autonomous agent that completes missions in a simple tactical scenario. The agent is designed with flexibility in mind. It reasons from first principles about high-level tactical decisions, and is thus able to reason in unexpected situations and recover gracefully from mistakes. In addition, the agent's knowledge base is flexible enough to be easily transferred between simulation platforms and to encode new tactics in a modular fashion. We are currently implementing the knowledge necessary for the agent to complete the 1-v-1 aggressive bogey scenario. This includes addressing the two important issues of situation interpretation and achieving multiple simultaneous and interacting goals.

Our future research will involve incrementally expanding the agent's knowledge base so it can reason robustly in a wide range of 1-v-1 scenarios. We will also soon focus on modeling more complex scenarios, including those involving more than two planes. This will also allow us to expand the agent's coverage of the cognitive behaviors involved in tactical flight. For example, we will incorporate more intelligent methods for situation assessment, modeling other agents (i.e., robustly predicting actions and goals of other participants in the scenario, both friends and foes), identifying potential threats, and reacting to them. Beyond that, we will focus on more complex cognitive tasks, such as more complete integration of planning, reaction, and execution, more sophisticated interpretation of the environment and other agents, and learning from instruction.

References

Covrigaru, A. (1992). Emergence of metalevel control in multi-tasking autonomous agents (Technical Report No. CSE-TR-138-92). Doctoral dissertation, Dept. of Electrical Engineering and Computer Science, University of Michigan.

Kornell, J. (1987). Reflections on using knowledge based systems for military simulation. *Simulation, 48*, 144–148.

Newell, A. (1990). *Unified theories of cognition.* Cambridge, MA: Harvard University Press.

Pearson, D. J., Huffman, S. B., Willis, M. B., Laird, J. E., & Jones, R. M. (1993). Intelligent multi-level control in a highly reactive domain. In *Proceedings of the International Conference on Intelligent Autonomous Systems.*

Ritter, F., & Feurzeig, W. (1987). Teaching real time tactical thinking. In J. Psotka, L. D. Massey, & S. A. Mutter (Eds.), *Intelligent tutoring systems: Lessons learned.* Hillsdale, NJ: Lawrence Erlbaum.

Rosenbloom, P. S., Laird, J. E., Newell, A., & McCarl, R. (1991). A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence, 47*, 289–325.

Zytkow, J. M., & Erickson, M. D. (1987). Tactical manager in a simulated environment. In Z. W. Ras & M. Zemankova (Eds.), *Methodologies for intelligent systems.* Amsterdam: Elsevier Science.

Intelligent Automated Agents for Tactical Air Simulation: A Progress Report

Paul S. Rosenbloom,[1] W. Lewis Johnson,[1] Randolph M. Jones,[2] Frank Koss,[2] John E. Laird,[2]
Jill Fain Lehman,[3] Robert Rubinoff,[3] Karl B. Schwamb,[1] and Milind Tambe[1]

[1]Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
{rosenblo, johnson, schwamb, tambe}@isi.edu

[2]Artificial Intelligence Laboratory
The University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109
{rjones, koss, laird}@eecs.umich.edu

[3]Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{jef, rubinoff}@cs.cmu.edu

Abstract

This article reports on recent progress in the development of TacAir-Soar, an intelligent automated agent for tactical air simulation. This includes progress in expanding the agent's coverage of the tactical air domain, progress in enhancing the quality of the agent's behavior, and progress in building an infrastructure for research and development in this area.

Introduction

At the Third Conference on Computer Generated Forces and Behavioral Representation we presented an initial report on an effort to build intelligent automated agents for tactical air simulation (Jones *et al*, 1993). The ultimate intent behind this effort is to develop automated pilots whose behavior in simulated battlefields is nearly indistinguishable from that of human pilots (and to go beyond this to develop generic agents that are readily specializable for this and other domains). If such agents can be created, they should provide close to ideal force supplements for many of the applications anticipated for distributed interactive battlefield simulation.

As of the initial report, prototype agents had been constructed that could, in real-time, flexibly use a small amount of tactical knowledge about two classes of one-versus-one (1-v-1) Beyond Visual Range (BVR) tactical air scenarios. In the *non-jinking bogey* scenarios, one plane (the non-jinking bogey) is unarmed and maintains a straight-and-level flight path. The other plane is armed with long-range radar-guided, medium-range radar-guided, and short-range infrared-guided missiles. Its task is to set up for a sequence of missile shots, at increasingly shorter ranges, until the non-jinking bogey is destroyed. Though such scenarios are not common in the real world, they are used as training exercises because they teach pilots how to position their planes for later shots while simultaneously taking earlier ones. In the *aggressive bogey* scenarios, one plane is attempting to protect a High-Value Unit (HVU), such as an aircraft carrier, via a Barrier Combat Air Patrol (BARCAP); that is, the plane patrols between the HVU and the anticipated threat (by cycling around a racetrack pattern), and then intercepts any threat that it detects in its sector. The other plane is attempting to attack the HVU, but to do so it must first intercept the defensive aircraft.

The prototype agents were all implemented as parameterized variations of a single multi-functional tactical-air agent, called *TacAir-Soar* (or *TAS* for short). TAS is built within *Soar*, a software architecture that is being developed as a basis for both an integrated intelligent system and a unified theory of human cognition (Rosenbloom, Laird, & Newell, 1993; Newell, 1990). Soar provides TAS with basic support for knowledge representation, problem solving, reactivity, external interaction, and learning. Soar also provides a potential means of integrating into TAS additional planning, learning, and natural language capabilities that are being developed independently within Soar.

The prototype TAS agents actually utilized only a subset of the capabilities provided either directly by Soar, or built separately within it. However, this subset – along with the domain-specific (and domain-independent) rules that were added to Soar's long-term memory – was sufficient to yield a combination of knowledge-based decision making, task(/goal) switching and decomposition, and real-time interaction with the DIS environment. Knowledge-based decision-making arises from Soar's ability to make decisions based on integrating *preferences* generated by arbitrary sets of rules. Task switching also arises from Soar's decision-making abilities, but here as specifically applied to the selection and switching of tasks. Tasks(/goals) are represented as *operators* in Soar, and are one of the main foci of its decision making. Task decomposition arises from using the same decision mechanism to drive task performance, plus Soar's ability to automatically generate a new performance context when a decision is problematic. When these mechanisms are combined with rules that generate preferences about which subtasks are appropriate for which problematic parent tasks (in the particular situation of interest), task decomposition occurs. Real-time interaction with the DIS environment arises from the combination of Soar's incorporation of perception and action within the inner loop of its decision making capabilities – thus allowing all decisions to be informed by the current situation (and interpretations of it, as generated by rule firings) – and the use of ModSAF (Calder *et al*, 1993) as the interface to the DIS environment (Schwamb, Koss, & Keirsey, 1994).

When combined with the very preliminary domain knowledge that was encoded at the time, this combination of capabilities yielded competent behavior for the non-jinking bogey scenarios, but only fragments of behavior for the aggressive bogey scenarios (due to insufficient knowledge about this class of scenarios). One type of aircraft, similar to an F14, was flown in these scenarios.

The purpose of this article is to provide a report, one year later, on the progress in moving TAS from the initial prototype agents towards the ultimate goal of human-like automated pilots that are broadly capable in tactical air scenarios. This report is intended to be complemented by the more detailed articles about particular aspects of TAS that also appear in these proceedings (Johnson, 1994a; Jones & Laird, 1994; Jones *et al*, 1994; Koss & Lehman, 1994; Laird & Jones, 1994; Rubinoff & Lehman, 1994; Schwamb, Koss, & Keirsey, 1994; Tambe & Rosenbloom, 1994; van Lent & Wray, 1994), rather than to substitute for them. Thus, where there is a potential overlap between this report and any of the more detailed articles, this report will become more terse and defer (and refer) to the appropriate detailed article(s).

In the body of this report, progress on *domain capabilities* will be covered first. The focus here is on expanding the classes of domain scenarios in which the agents can behave appropriately. Second, progress on *intelligent capabilities* will be covered. The focus here is on expanding the classes of basic intelligent abilities – such as coping with multiple interacting tasks, plan recognition, learning, planning, self-explanation, and natural language – exhibited by the agents. Third, progress on *infrastructure capabilities* – such as integration with the DIS simulation environment, low-cost interfaces for human pilots, knowledge acquisition, and documentation – will be covered. Finally, the article will be concluded with plans for the future.

Domain Capabilities

Progress on domain capabilities has occurred in two general areas: (1) improving the robustness and range of the scenarios within 1-v-1 BVR tactical air; and (2) scaling up the scenarios in terms of the number of vehicles, the range of vehicle types, and the complexity of the required organization and communication among the vehicles.

Within 1-v-1, the TAS agents can now exhibit competent behavior in the BVR tactical-air segments of the aggressive bogey scenarios. This includes the ability to patrol in a racetrack pattern; select radar modes, detect opponents on radar, perform search and acquire activities when opponents drop off of radar, and maneuver so as to confuse the opponent's search and acquire activities; determine and attempt to achieve appropriate intercept geometries and launch-acceptability regions (LARs); select, fire, and support missiles; and detect and evade enemy missiles.

As played out in the DIS environment, a typical aggressive bogey scenario involves an F14 which is defending its aircraft carrier against possible attack by a MiG29. The F14 patrols in a racetrack until it spots the MiG29 (the F14's radar and missiles both have longer ranges than do the MiG29's). The F14 continues to monitor the MiG29 until its commit criteria are achieved, at which point it begins the intercept by attempting to achieve a good geometry from which to fire a long-range missile (LRM). At some later point the MiG29 detects the F14 and also then begins an intercept. This makes it difficult for the F14 to achieve any further advantage in intercept geometry, so it gives up on that, and turns to maximize the rate of closure (and thus to minimize the time before the intercept is complete).

When the F14 is finally close enough (that is, it has the MiG29 within its LRM's *launch-acceptability region*), and is oriented correctly, it launches a long-range missile, and performs an *fpole* (a turn that decreases the rate of closure between the aircraft – to delay the arrival of any missiles that might have been launched from the MiG29 – while simultaneously keeping the MiG29 on the F14's radar). The MiG29 detects the fpole, and *beams* in response, by turning perpendicular to the F14 (to render blind the Doppler radar that is guiding the F14's missile). The F14 then attempts to search for and reacquire the MiG29, while simultaneously changing altitude in order to confuse the MiG29's search and acquire activities.

Both planes then generally attempt to set up for further missile launches, and to avoid missiles launched by their opponents. Depending on the exact timing of the engagement, and on the willingness of the two planes to take risks (this is a TAS parameter), zero, one, or both of the planes may be shot down in the process.[1]

This scenario can be played out with both planes flown by TAS agents, or with one or the other flown by a human pilot in a flight simulator. A formal demonstration of the aggressive bogey scenario in the WISSARD laboratory at Oceana Naval Air Station during June '93 successfully pitted two TAS agents in simulated F14s against two human pilots (in F18 simulators, but acting as MiG29s). This demonstration was set up as two independent 1-v-1 engagements (out of radar range of each other). Given the early state of development of the agents at the time, the human pilots were constrained in terms of the kinds of

[1]In real engagements, if one or more of the aircraft survive the BVR segment of the scenario, either a within-visual range (WVR) engagement – that is, a dogfight – or an air-to-ground attack on the HVU may then occur. However, these aspects of the scenario are not part of BVR tactical air, and are thus not pursued by the TAS agents.

tactics they were allowed to use. Under these circumstances the demonstration proceeded successfully, in real-time, and in an otherwise unscripted manner. The resulting behavior was much as described in the typical example above. Feedback from Navy personnel in attendance at the demonstration was uniformly positive.

Despite this demonstrable success – and the fact that in numerous subsequent presentations to domain experts and other Navy personnel TAS has consistently impressed with its quality of behavior – it must be noted that TAS is still not close to covering the full complexity of the domain abilities described above, or the interactions among them. For example, only a subset of the radar modes are used; search and acquire in three dimensions is not strong; and only a subset of the possible tactics for patrolling, confusing, intercepting, and evading are used. Fleshing out these abilities does not look conceptually difficult at this point, just time consuming.

Another dimension of complexity in 1-v-1 BVR tactical air that is not fully addressed at this point by TAS is the space of possible missions that the agents need to be able to perform. The aggressive bogey scenarios cover two types of missions (BARCAP-HVU and ATTACK-HVU); however, there is still a handful of others. One other mission to which TAS has recently been extended is a MiGSWEEP. A MiGSWEEP is a sweep by one side's fighters through the other side's territory to clear out a corridor for later aircraft (such as bombers). In addition to the abilities required for the previous missions, a MiGSWEEP requires the ability to fly to waypoints, and to break off an intercept and "blow through" an opponent (that is, engage in a small amount of WVR behavior in order to accomplish a high-speed pass of an opponent and continue with the planned flight path).

In scaling up from these 1-v-1 scenarios, the biggest change has been the incorporation of an ability to detect and manage multiple aircraft. In 2-v-1 engagements a *section* (i.e., a coordinated pair of planes) must be able to fly together in formation and execute coordinated tactics. In service of this they must be able to communicate with each other, and to be aware of each other's positions. The TAS agent is now capable of doing this (as discussed in the next section) to support competent 2-v-1 behavior, within the same kinds of limits described for 1-v-1 (Jones & Laird, 1994; Laird & Jones, 1994).

In 1-v-2 engagements a single aircraft must be able to identify and sort out the activities of a pair of adversaries who may or may not be flying together as a section (Jones & Laird, 1994). It must be able to work out intercept geometries that take both opponents into account – so as, for example, not to be sandwiched between them. It must also be able to determine which of the pair is the primary threat, target the primary threat, and determine when to also fire at the secondary threat. For example, if the pair are flying in a coordinated fashion, then firing a missile at one is likely to cause both to beam. It would thus be a waste to launch missiles at both under such circumstances. The current TAS agents are also capable of performing competently in such 1-v-2 engagements.

In 2-v-2 engagements, many of the same issues come up as in 1-v-2 and 2-v-1. However, additional capabilities are required to *sort* the opponents (determining which friendly aircraft has the responsibility for which opponent aircraft), to decide when one or both aircraft should launch missiles, and to decide when to split the single 2-v-2 engagement into two independent 1-v-1 engagements (i.e., to *strip*). Though work on 2-v-2 has just recently begun, there is now at least one working example of a section of TAS agents successfully sorting and firing at another section of TAS agents. Once 2-v-2 is completed, larger engagements (2-v-N, 4-v-4, N-v-N, etc.) will still remain to be covered.

Other aspects of scale up that are currently in progress include adding the ability to fly an F18 (to the original F14 and the recently added MiG29), and the addition of an air intercept control (AIC) agent in an E2 (a specialized radar plane that is similar to an AWACS) (Rubinoff & Lehman, 1994). The AIC's job is different in a number of ways from that of a fighter pilot, so stretching TAS to accommodate this new type of agent should force further generalization of its capabilities.

Intelligent Capabilities

With respect to intelligent capabilities, the most significant advance over the prototype agents has been the addition to TAS of the ability to maintain episodic memories of its engagements, and to use these memories in reconstructing what it did, why it did what it did, and what else it would have done if the situation had been slightly different (Johnson, 1994a; Johnson, 1994b). These description and explanation capabilities are available through an interactive debriefing interface, in which questions can be asked via selection from dynamically created menus, and answers are generated in (approximate) English. In contrast to explanation in most expert systems, where there is a distinct "explanation" system that has direct access to the "performance" system's knowledge and derivational traces, TAS generates the explanations itself based only on (1) what it can remember about what happened and (2) what it can later reconstruct about what it might have done (and why it might have done it). This is a process that can be misled by circumstances, but it is expected to be more like how human pilots would actually describe and explain their own behavior during post-mission debriefing (though the psychological and behavioral accuracy of this has not yet been studied).

In addition to these debriefing capabilities, significant progress has also been made on incorporating several other capabilities into TAS. One capability is coping with multiple interacting goals. Though mapping a forest of interacting goals onto the single goal stack maintained by Soar has turned out to be non-trivial – and is currently a topic of intensive investigation – workable strategies have been found for TAS agents to coordinate their behavior in the presence of all of these goals and their interactions (Jones *et al*, 1994). A second capability is integrating information from multiple sources about multiple agents (Jones & Laird, 1994). The sources of information about other agents have been expanded from just radar, to also include radio and vision;[2] and the number of agents about which information can be represented has been expanded from one up to an arbitrary number. A third capability is communication and coordination among multiple agents (Laird & Jones, 1994). Instead of modeling a group of related agents – such as a section of aircraft or a platoon of tanks – as a single aggregate unit, the behavior of groups is being modeled at the individual platform level. This provides additional flexibility and realism in the simulation, but also necessitates modeling how the groups actually do communicate and coordinate among themselves.

Additional capability investigations are also underway in the areas of learning, planning, plan recognition and natural language. Learning and planning are a relatively common part of Soar's repertoire of behaviors in general (Laird & Rosenbloom, 1990); however, they are not yet a routine part of TAS's behavior. Investigations of their use in TAS have begun – for example, the debriefing capability depends on learning being active within certain key portions of the TAS agents – but it is too early to comment generally on their outcome. In contrast, plan recognition is now a routine part of TAS's

[2]The radar, vision, and radio inputs attempt to provide TAS with the information a human would extract from those sources. However, this information is provided symbolically, and no actual visual or audio processing on the part of the agent is required.

behavior, but only of a simple, low-level, ad hoc variety. For example, when an opponent turns, a new (hand-coded) task may be selected to interpret whether the opponent is performing an fpole (as part of a missile launching plan) or a beam (as part of a missile evasion plan). General plan recognition turns out to be particularly difficult in the DIS environment because of the presence of partial information about multiple, flexible, interacting agents. However, a more systematic approach based on abstract *model tracing* (Anderson *et al*, 1990; Ward, 1991) in (multi-agent) world-centered models is being investigated in a version of TAS, and is showing some promise (Tambe & Rosenbloom, 1994). Finally, an investigation is in progress on how to incorporate independently developed, Soar-based, natural-language abilities (Lehman, Lewis, & Newell, 1991) into TAS (Rubinoff & Lehman, 1994). In theory, two automated agents could communicate without using natural language; however, to do so can affect how they are perceived by agents that are eavesdropping on them. In the longer run, natural language is also a critical capability if automated agents are ever to interact in a seamless way with human agents. Natural language communication will initially be provided between a pair of TAS agents – a fighter and an AIC (in an E2) – with further deployment hopefully to follow.

Infrastructure Capabilities

With respect to infrastructure, progress has been made on four topics: (1) integration of Soar with the DIS simulation environment; (2) provision of a low-cost interface for human pilots; (3) knowledge acquisition methodology; and (4) documentation tools and methodology. These topics are covered in turn here.

TAS agents are now able to act as full participants within the DIS battlefield simulation environment. The key to this has been the use of ModSAF 1.0 (Calder *et al*, 1993) as an intermediary between Soar and DIS (Schwamb, Koss, & Keirsey, 1994). ModSAF already contains an interface to DIS, so it was only necessary to add an interface between Soar and ModSAF. To do this we have implemented a *cockpit abstraction* on top of ModSAF that allows TAS to focus on behaving like a pilot, while ModSAF simulates vehicles, sensors, and weapons. TAS is not utilizing ModSAF's own pilot behaviors (such as Sweep, CAP, and Fly Route), as programmed into its tasks and task frames; however, TAS's piloting task has been simplified somewhat by providing it high-level flight control via a ModSAF library function that accepts as parameters a desired altitude, heading, etc. In addition to adding the cockpit abstraction (and getting Soar to use it), we have extended the implementation of Soar so as to allow multiple independent Soar agents within a single process. This has allowed multiple TAS agents to be compiled together with ModSAF in a single process,[3] and thus allowed communication between the agents and ModSAF to be mediated directly by calling library functions (rather than through slower interprocess communication mechanisms, such as sockets).

Given a cockpit abstraction, it turned out to be relatively easy to reuse it in support of a low-cost interface for human control of ModSAF aircraft. The Human Instrument Panel (HIP) provides an X-Windows-based interface to a vehicle's cockpit abstraction (van Lent & Wray, 1994). This enables a human pilot to perceive graphically-presented sensor information and to control the aircraft's flight, weapons, and sensors at the same level at which they are controlled by TAS agents. Easily being able to control ModSAF vehicles at this level of detail, and on any workstation, has proven to be quite useful in testing and experimenting with

[3]Soar is currently implemented in C – as is ModSAF – without which this integration would have been considerably more difficult.

TAS agents. However, the HIP clearly can't completely replace the functionality of higher fidelity (and cost) flight simulators.

With respect to knowledge acquisition, the most important development has been the opening of the WISSARD laboratory at Oceana Naval Air Station (in Norfolk, VA). The lab contains two high fidelity (dome) aircraft simulators; two medium fidelity aircraft simulators; plus workstations for running ModSAF, TAS, and several visualization and analysis tools. The laboratory has enabled us to add to the standard knowledge acquisition methodologies the ability to watch, tape, and log, engagements among human pilots (both official "subject matter experts", as well as operational pilots), and engagements between human pilots and TAS agents.

With respect to documentation, we have developed substantial portions of a three layer hypertext document that links together: (1) knowledge about the domain (as extracted from books, experts, etc.); (2) a description of the structure and content of TAS; and (3) the actual rules that comprise TAS (Koss & Lehman, 1994). This documentation has been developed within NCSA Mosaic, a distributed, multi-media, hypertext system. It is expected to facilitate understanding and validation of the knowledge and code embodied in the automated agents.

Summary and Future

TAS is now capable of performing competently in beyond-visual range tactical-air scenarios containing up to three interacting aircraft. Moreover, it can do so while flying two types of aircraft in service of three types of missions. It can also participate in interactive post-mission debriefings about its engagements.

These various capabilities arise from combining knowledge about the tactical air domain with a set of "intelligent" abilities embodied by TAS for knowledge-based decision making, reactive real-time interaction, coping with multiple interacting goals, integrating information from multiple sources about multiple agents, communication and coordination, episodic memory, and reconstructive self-description and self-explanation.

The basic TAS agent is coded within Soar via 145 operators, where each operator corresponds to a task (or goal) at some level of granularity. In terms of rules, the implementation involves approximately 1,500. Most of these rules are responsible for proposing, selecting, and applying the operators, but some do perform other tasks (such as encoding perceptual input, and elaborating state descriptions). The debriefing capability adds another 80 operators, amounting also to approximately 1,500 rules. So the combined system consists of 225 operators and approximately 3,000 rules. The natural language capabilities that are currently being added utilize an additional 56 operators, and approximately 900 rules. Note that these operator and rule counts are all "before learning", as learning can increase both the number of rules and the number of operators.

Beyond the agent itself, progress has also been made on building an infrastructure to support research and development on intelligent automated agents for tactical air, and beyond.

Plans for the coming year include completing 2v2 BVR tactical air, and transitioning TAS from tactical air to *close air support* (a form of air-to-ground engagement). We also expect to have planning, learning, and plan recognition working routinely in TAS, and to have limited amounts of natural language also in routine use. Meanwhile, incremental improvements are expected to continue on the infrastructure for research and development.

Acknowledgment

This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan, as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). Critical support has been provided by Dennis McBride of ARPA/ASTO; Tom Brandt, Bob Richards, and Ed Harvey of BMH Inc.; Andy Ceranowicz and Joshua Smith of Loral Inc.; and David Keirsey of the Hughes Aircraft Co.

References

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. 1990. Cognitive modeling and intelligent tutoring. *Artificial Intelligence* , *42*, 7-49.

Calder, R. B., Smith, J. E., Courtemanche, A. J., Mar, J. M. F., & Ceranowicz, A. Z. 1993. ModSAF behavior simulation and control. *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation.* Orlando, FL: Institute for Simulation & Training. pp. 347-356.

Johnson, W. L. 1994. Agents that explain their own actions. *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL.

Johnson, W. L. 1994. Agents that learn to explain themselves. *Proceedings of the Twelfth National Conference on Artificial Intelligence*. Seattle: AAAI, In press.

Jones, R. M. & Laird, J. E. 1994. Multiple information sources and multiple participants: Managing situational awareness in an autonomous agent. *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL.

Jones, R. M., Tambe, M., Laird, J. E., & Rosenbloom, P. S. 1993. Intelligent automated agents for flight training simulators. *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation.* Orlando, FL. pp. 33-42.

Jones, R. M., Laird, J. E., Tambe, M., & Rosenbloom, P. S. 1994. Generating behavior in response to interacting goals. *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL.

Koss, F. & Lehman, J. F. 1994. Knowledge acquisition and knowledge use in a distributed IFOR project. *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL.

Laird, J. E. & Jones, R. M. 1994. Coordinated behavior of computer generated forces in TacAir-Soar. *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL.

Laird, J. E., & Rosenbloom, P. S. 1990. Integrating execution, planning, and learning in Soar for external environments. *Proceedings of the Eighth National Conference on Artificial Intelligence*. Boston: AAAI, MIT Press. pp. 1022-1029.

Lehman, J. F., Lewis, R. L., & Newell, A. 1991. Integrating knowledge sources in language comprehension. *Proceedings of the Thirteenth Annual Meeting of the Cognitive Science Society*. Hillsdale, NJ, Erlbaum. pp. 461-466.

Newell, A. 1990. *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (Eds.) 1993. *The Soar Papers: Research on Integrated Intelligence*. Cambridge, MA: MIT Press.

Rubinoff, R. & Lehman, J. F. 1994. Natural language processing in an IFOR pilot. *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL.

Schwamb, K. B., Koss, F. V., & Keirsey, D. 1994. Working with ModSAF: Interfaces for programs and users. *Proceedings of* *the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL.

Tambe, M. & Rosenbloom, P. S. 1994. Event tracking in complex multi-agent environments. *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL.

van Lent, M. & Wray, R. 1994. A very low cost system for direct human control of simulated vehicles. *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL.

Ward, B. May 1991. *ET-Soar: Toward an ITS for Theory-Based Representations*. Doctoral dissertation, School of Computer Science, Carnegie Mellon University,

Biographies

Paul S. Rosenbloom is an associate professor of computer science at the University of Southern California and the acting deputy director of the Intelligent Systems Division at the Information Sciences Institute. He received his B.S. degree in mathematical sciences from Stanford University in 1976 and his M.S. and Ph.D. degrees in computer science from Carnegie-Mellon University in 1978 and 1983, respectively. His research centers on integrated intelligent systems (in particular, Soar), but also covers other areas such as machine learning, production systems, planning, and cognitive modeling. He is a Councillor of the AAAI and a past Chair of ACM SIGART.

W. Lewis Johnson is a project leader at the University of Southern California Information Sciences Institute, and a research assistant professor in the USC Department of Computer Science. Dr. Johnson received his A.B. degree in linguistics in 1978 from Princeton University, and his M.Phil. and Ph.D. degrees in computer science from Yale University in 1980 and 1985, respectively. He is interested in applying artificial intelligence techniques in the areas of computer-based training and software engineering. His current projects are developing tools that automate the generation of software documentation, and that explain the problem solving of intelligent agents.

Randolph M. Jones received his Ph.D. in information and computer science from the University of California, Irvine, in 1989. He is currently an assistant research scientist in the Artificial Intelligence Laboratory at the University of Michigan. His primary research interests lie in the areas of intelligent agents, problem solving, machine learning, and psychological modeling.

Frank V. Koss is a systems research programmer in the Artificial Intelligence Laboratory at the University of Michigan, where he is developing the interface between the Soar architecture and the ModSAF simulator. He received his BS in computer engineering from Carnegie Mellon University in 1991 and his MSE in computer science and engineering from the University of Michigan in 1993. He is a member of IEEE and AAAI.

John E. Laird is an associate professor of electrical engineering and computer science and the director of the Artificial Intelligence Laboratory at the University of Michigan. He received his B.S. degree in computer and communication sciences from the University of Michigan in 1975 and his M.S. and Ph.D. degrees in computer science from Carnegie Mellon University in 1978 and 1983, respectively. His interests are centered on creating integrated intelligent agents (using the Soar architecture), leading to research in problem solving, complex behavior representation, machine learning, cognitive modeling.

Jill Fain Lehman is a research computer scientist in Carnegie Mellon's School of Computer Science. She received her B.S. from Yale in 1981, and her M.S. and Ph:D. from Carnegie Mellon in 1987 and 1989, respectively. Her research interests span the area of natural language processing:

comprehension and generation, models of linguistic performance, and machine learning techniques for language acquisition. Her main project is NL-Soar, the natural language effort within the Soar project.

Robert Rubinoff is a postdoctoral research fellow in Carnegie Mellon's School of Computer Science. He received his B.A., M.S.E., and Ph.D. from the University of Pennsylvania in 1982, 1986, and 1992, respectively; his dissertation research was on "Negotiation, Feedback, and Perspective within Natural Language Generation". His research interests include natural language processing, knowledge representation, and reasoning. He is currently working on natural language generation within the Soar project.

Karl B. Schwamb is a Senior Programmer Analyst on the Soar Intelligent FORces project at the University of Southern California's Information Sciences Institute. He is primarily responsible for the maintenance of the Soar/ModSAF interface software described in this article. He received his M.S. in Computer Science from George Washington University.

Milind Tambe is a computer scientist at the Information Sciences Institute, University of Southern California (USC) and a research assistant professor with the computer science department at USC. He completed his undergraduate education in computer science from the Birla Institute of Technology and Science, Pilani, India in 1986. He received his Ph.D. in 1991 from the School of Computer Science at Carnegie Mellon University, where he continued as a research associate until 1993. His interests are in the areas of integrated AI systems, and efficiency and scalability of AI programs, especially rule-based systems.

Agents that Explain Their Own Actions

W. Lewis Johnson
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
johnson@isi.edu

Abstract

Computer generated battlefield agents need to be able to explain the rationales for their actions. Such explanations make it easier to validate agent behavior, and can enhance the effectiveness of the agents as training devices. This paper describes an explanation capability called Debrief that enables agents implemented in Soar to describe and justify their decisions. Debrief determines the motivation for decisions by recalling the context in which decisions were made, and determining what factors were critical to those decisions. In the process Debrief learns to recognize similar situations where the same decision would be made for the same reasons. Debrief currently being used by the TacAir-Soar tactical air agent to explain its actions, and is being evaluated for incorporation into other reactive planning agents.

1 Introduction

The Soar-IFOR project [15] is developing intelligent agents that can control tactical aircraft in distributed battlefield simulations. A key objective of the project is to endow such simulated agents with human-like behavior. Human players in the simulation should not be able to tell which units are controlled by humans and which are controlled by computer, lest they come to rely on this knowledge. Simulations can serve as an effective test bed for development and evaluation of tactics only if the agents realistically employ those tactics.

Yet it is difficult to validate through observation that agent behavior really is human-like. Behavior depends upon the agent's goals and situation assessments, and these can change from moment to moment. A given action might be appropriate in one situation, and altogether inappropriate in a slightly different situation. Therefore the fact that an agent happens to behave realistically in one scenario is no guarantee that the agent will perform properly in other scenarios. In order to gain confidence in the accuracy of the agent's behavior it is helpful to examine its reasoning processes, and compare them against human reasoning.

In order to produce human-like behavior we have focussed on modeling human thought processes and reasoning, using the Soar cognitive architecture [14]. These thought processes are made visible using an explanation capability, called Debrief, that describes and answers questions about the agent's actions and decisions. Debrief can also point out alternative actions that might have been taken, but were rejected. This helps to ensure that the actions were performed for the right reasons, and were not chance occurrences.

Debrief was inspired by post-flight debriefings in tactical training. Debriefings are conducted after training exercises so that instructors and trainees can understand what went wrong and why, and draw lessons that can be applied to future engagements. Similar capabilities in Debrief make it easier for people to understand and improve the performance of simulated agents.

2 Objectives and Approach

The following are the design objectives for Debrief in TacAir-Soar.

1. It should describe an engagement from the agent's perspective, explaining what the agent's objectives where, what actions it took to meet those objectives, and its assessment of the unfolding situation.
2. It should accept follow-on questions about those actions, objectives, and assessments, justifying them as appropriate.

3. In explaining actions it should use a combination of media familiar to potential users, including natural language and diagrams.

4. These capabilities should be provided without unnecessary impact on the design and implementation of other agent capabilities.

5. The explanation capability should not be specific to tactical air-to-air combat, but should be applicable to a variety of domains.

The most obvious way to provide such explanations would be to generate English paraphrases of the rules and rule firing traces used by TacAir-Soar. Yet such techniques have proved to be ineffective for explaining expert system reasoning [4, 16, 3], and are likely to be inappropriate in computer generated forces as well. They contain too many implementation details, and are too dependent upon the particulars of how knowledge is encoded in the system. It is necessary to abstract away from these details, and focus on the essential knowledge underlying the agent's decisions.

Another approach might be to encode the underlying knowledge explicitly, either in a declarative form or as abstract meta-rules [2]. Such an approach has a number of potential problems. First of all, computer generated forces require a great variety of reasoning capabilities, including planning, plan recognition, learning, and geometric reasoning. The problem solving strategies and domain knowledge representations required for many of these capabilities are matters of current research. If such knowledge were encoded declaratively, but the agent does not make direct use of it, and instead employs procedural rules or codes, the declarative descriptions will quickly become out of date as the agent is extended and modified. This is especially true for experimental intelligent systems like TacAir-Soar. Yet if it did reason directly from declarative representations its performance would suffer, and the design of the agent would be greatly affected, in contradiction to point 4 listed above. Automated compilation of declarative knowledge [13] can help eliminate the performance problems, but compilation techniques have not yet been employed in systems that have have as great a variety of reasoning capabilities as TacAir-Soar does.

Debrief takes a fundamentally different approach to explanation. In order to determine the rationales for an agent's decision, it recalls the situation in which the decision was taken, and then replays the agent's tactical reasoning processes. By monitoring the reasoning, and experimenting with changing the situation in various ways, it discovers the critical factors in the situation that led to the decision. These results are learned so that they can be applied to other decisions in similar situations. In effect the agent constructs a declarative model of its own reasoning through reflection. This is analogous to human explanation generation—experts often can perform tasks without being conscious of the rationales underlying their performance, and must reflect afterwards to determine what those rationales might have been.

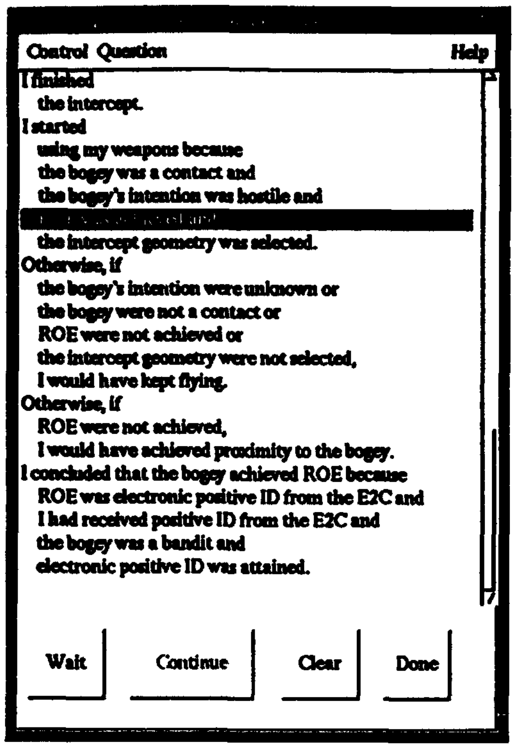
Although Debrief was originally intended for explanation in the tactical air-to-air domain, the implementation is not specific to that domain. TacAir-Soar is in the process of being adapted to handle air-to-ground operations; it is expected that these changes will little or no impact on Debrief. Plans are underway to apply Debrief to an entirely different domain, namely automated control of radar tracking stations in the NASA Deep Space Network [8, 7].

3 An Example

The following example scenario illustrates how Debrief is employed. Suppose that the TacAir-Soar agent is assigned a Barrier Combat Air Patrol (BARCAP) mission, i.e., to search the skies for enemy aircraft and intercept them so that they cannot threaten a high value unit such as an aircraft carrier. During the course of the mission the agent detects a hostile aircraft. The agent intercepts the aircraft, fires a missile at it which destroys it, and then resumes its patrol.

After the engagement Debrief can be used to ask the agent questions. Dialog is conducted via a menu-oriented interface. First, the user requests that Debrief describe what took place during the engagement; it then gives a step-by-step description of the mission. If there is any statement in the description that the user has a question about, he or she can button on it with the mouse and request an elaboration or explanation.

Figure 1 shows part of the display during the course of the interaction. The user has selected one of the statements in the description, "I started using my weapons," and has requested a justification for that decision. The explanation for this step is displayed in the window shown; the original description of the mission has almost entirely scrolled off the top of the window. It lists sev-



```
I finished
  the intercept.
I started
  using my weapons because
  the bogey was a contact and
  the bogey's intention was hostile and
  ROE was achieved and
  the intercept geometry was selected.
Otherwise, if
  the bogey's intention were unknown or
  the bogey were not a contact or
  ROE were not achieved or
  the intercept geometry were not selected,
  I would have kept flying.
Otherwise, if
  ROE were not achieved,
  I would have achieved proximity to the bogey.
I concluded that the bogey achieved ROE because
  ROE was electronic positive ID from the E2C and
  I had received positive ID from the E2C and
  the bogey was a bandit and
  electronic positive ID was attained.
```

Figure 1: The Debrief interaction window

eral reasons why the agent elected to use weapons against the bogey: the agent had radar contact with the bogey, it was known to be hostile, the rules of engagement (ROE) were satisfied, and the agent had already planned an intercept trajectory for closing in on the bogey. It also lists alternative actions that it might have taken but did not; for example, if ROE had not been satisfied, the agent would have closed in on the bogey, but would have refrained from firing weapons at it.

It is also possible to investigate why the agent reached particular conclusions during the course of the engagement. For example, since the conclusion that ROE was achieved was crucial to the agent's decision to employ weapons, it would be useful to find out why the agent reached this conclusion. This can be accomplished by selecting a conclusion and asking a follow-up question about it. In the window in the figure the conclusion "ROE was achieved" has been selected with the mouse, and an explanation for why this conclusion was made appears at the bottom of the figure.

4 Architecture of Debrief

Figure 2 shows the overall architecture of Debrief, and how Debrief fits into the architecture of TacAir-Soar. Like all Soar systems, TacAir-Soar is divided into *problem spaces*, each of which is responsible for a particular subtask. These problem spaces are organized hierarchically, where lower level problem spaces accomplish goals that are posed in the higher level spaces. The top level space performs the switch between principal modes of operation, namely accepting mission orders, flying the mission, and debriefing the mission.

In order to support debriefing, the mission problem spaces are augmented with an *event memory*, which is a record of the events that occurred during the mission. A set of operators and productions monitor TacAir-Soar's problem solving state during the execution of the mission in order to construct this memory. A separate *working memory specification* indicates which objects in Soar's internal working memory should be monitored for state changes. After the engagement Debrief retrieves information about the engagement from the event memory and uses it to describe and explain the mission. The working memory specification also helps to determine how to recall episodes from the event memory and analyze them and what information about those episodes is presented to the user.

The debriefing itself is performed within the Debrief problem space, which alternates between prompting the user for questions via the Prompt-for-Question problem space, and answering them via the Generate-Answer problem space. Generating answers involves recalling states in which events occurred, analyzing the rationales for the events or the beliefs that the agent held at the time, and then presenting the results to the user. The natural language generation capability within the presentation subsystem is also used in the Prompt-for-Question problem space to construct menus of events and decisions that the user may select from when forming a question.

The key aspects of each of these capabilities within Debrief will be described in more detail below.

5 Inputing Questions

Analysis of videotapes of mock post-flight debriefings indicates that a variety of types of questions can arise during the course of a debriefing. The question input capability is designed to enable users to pose many of these types of questions, without making users type their questions in En-

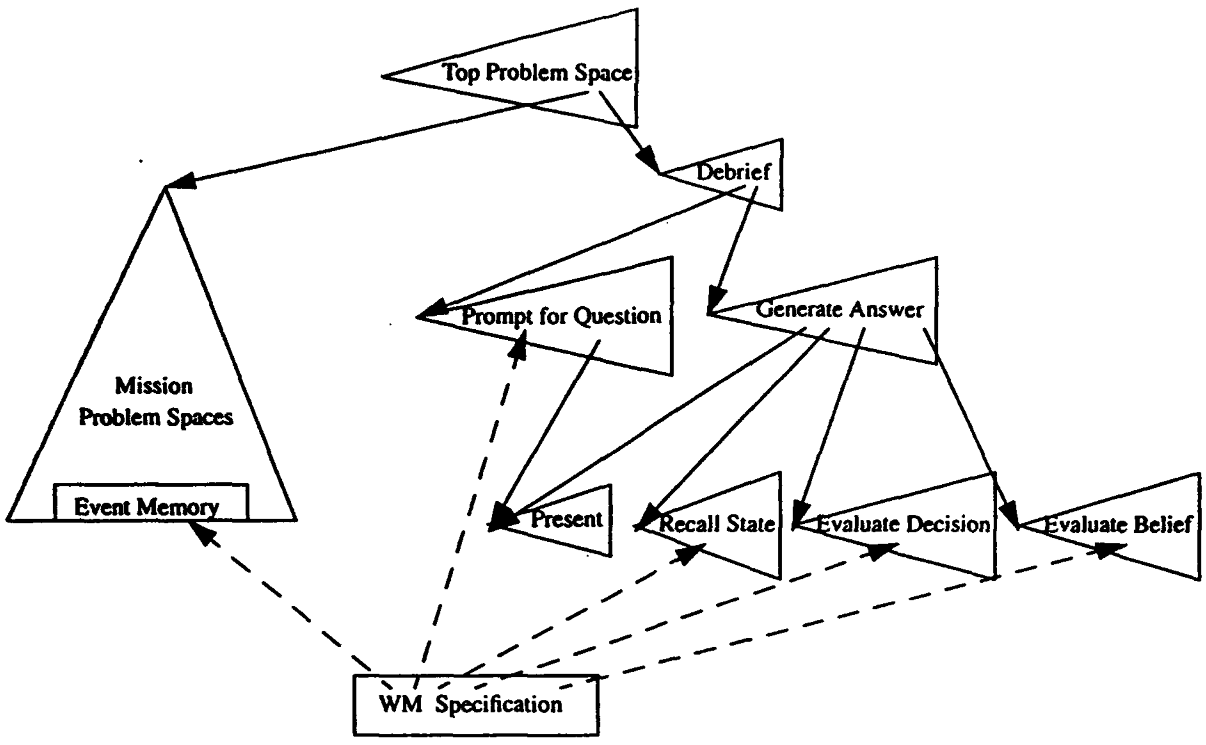


Figure 2: The architecture of Debrief within TacAir-Soar

glish and requiring Debrief to understand natural language input.

Questions were categorized into major semantic types, following the methodology that is common in question-answering systems [10]. Question types currently supported include: Describe-Event—describe an action or event and its circumstances; Explain-Action—explain why the agent performed a particular action; Explain-Conclusion—explain why the agent drew a particular conclusion; and Explain-Belief—explain why the agent believed that a particular fact was true. Instead of inferring the question type from the user's input, Debrief requires that the user explicitly select a question type from a menu. This avoids the problems of interpreting poorly articulated questions, but does require that the user understand the meaning of the question types, and to understand the distinction being made between actions and conclusions.

Each question applies to a specific event, decision, or belief. These can be selected via the interface. Initially when the user selects a question type Debrief lists the events in its event memory that are of the appropriate type for the question. The user may then select an element from this list. Subsequent questions can be formed in the same manner, or by selecting fragments of text with the mouse, using a technique similar to that employed by Moore and Swartout [12]. In the latter case the question is taken to refer to the event or belief described by that fragment of text.

6 Memory and Recall

Event memory in Debrief takes advantage of the fact that the major problem solving steps in Soar systems, namely problem spaces and operators, are represented explicitly in working memory. Events and decisions are recorded by productions that check for particular operators being applied. All major decisions within TacAir-Soar are implemented as operators, as are situation assessments. For example, the decision to use weapons is performed by an operator called Employ-Weapons, so a production was added to TacAir-Soar that fires whenever that operator is applied. An event token is then added to the event memory indicating the operator that was applied and the problem space in which it was applied.

Whenever an event is added to the event memory, TacAir-Soar's working memory is checked to see if there have been any changes since the last event. Soar's memory is organized as a collection of objects with attributes and values. Debrief monitors a subset of these attributes, and adds a record to the event memory whenever a change in values occurs.

The decision of what state information record depends upon what information is required during debriefing to explain decisions. Recording all state information would be costly in a complex system such as TacAir-Soar, and proves to be unnecessary. The working memory specification determines what information should be monitored and recorded. For each attribute of interest the specification indicates what types of values the attribute can take, whether or not an attribute can assume multiple values at once, and how those values change over time. If the values of the attributes are themselves complex objects with attributes, they are specified in the same way.

It was argued in Section 2 that duplicate representations of knowledge in declarative and procedural form can lead to maintenance problems. However, this is not a serious problem in the case of the working memory specification because the specification only describes the structure of working memory, not how that working memory is constructed and used. It is therefore relatively immune to modifications to the TacAir-Soar agent's rule base. Additionally, Debrief can optionally be made to check whenever the working memory state disagrees with the specification, and warn the developers that this is the case. The advantage is that Debrief can be incorporated into a new Soar system simply by identifying the operators in that system that are to be explained and constructing a specification for the system's working memory.

It is important that the event and state recording processes not add significantly to working memory size, since this could degrade the run-time performance of TacAir-Soar. Therefore Debrief makes use of Soar's learning mechanism, called chunking, in order to reduce working memory load. Chunking creates new production rules that Soar can then use in subsequent problem solving. Debrief builds so-called recognition chunks during the course of the engagement in order to facilitate the recall of state information. A recognition chunk will fire whenever a situation that Soar has encountered before arises again, enabling Soar to recognize that the situation has been encountered before. Debrief continually builds recognition chunks during the engagement, associating state changes with events. During debriefing the Recall-State problem space makes use of these chunks to reconstruct the state in which a given event occurs. It proposes a range of possible state changes, and if recognition chunks fire Debrief then knows that the state change was associated with the event. Once Recall-State has reconstructed the working memory state associated with an event a new chunk is built associating the event with the complete state description. Then if subsequent questions refer back to the same event the chunk leads to the immediate recall of the working memory state.

During the engagement TacAir-Soar's working memory size stays relatively constant, but its production memory constantly grows as new chunks are built. One might therefore be concerned that the additional productions would lead to decreased performance. Fortunately, studies have shown that Soar systems can be run with as many as a million chunks in them without signficant slowdown[5].[1] This number of chunks is far greater than Debrief has yet been required to produce.

7 Explaining Decisions and Beliefs

Once the circumstances surrounding a decision has been recalled, it is then possible for Debrief to determine what aspects of those circumstances led to the decision. Figure 3 shows the problem spaces that are involved in this process. The first step is to replay the original decision, to verify that circumstances leading to the decision have been correctly recalled. In essence Debrief is performing a kind of "what-if" simulation internal to Soar, checking to see what TacAir-Soar would do if it were in the recalled situation. Interaction between TacAir-Soar and ModSAF is disallowed, so that this what-if simulation does not have an unintended effect on the ModSAF vehicle that TacAir-Soar is controlling.

For example, in the case of the decision to use weapons described above, Debrief recalls the operator that was applied (Employ-Weapons), the problem space in which it was applied (Intercept), and the state in which it is applied (reconstructed by Recall-State). This information is passed to the Evaluate-Decision problem space, which in turn passes control to a problem space called Test Operator Applicability that sets up the Intercept problem space so that it can be reinvoked in the re-

[1] Robert Doorenbos, personal communication.

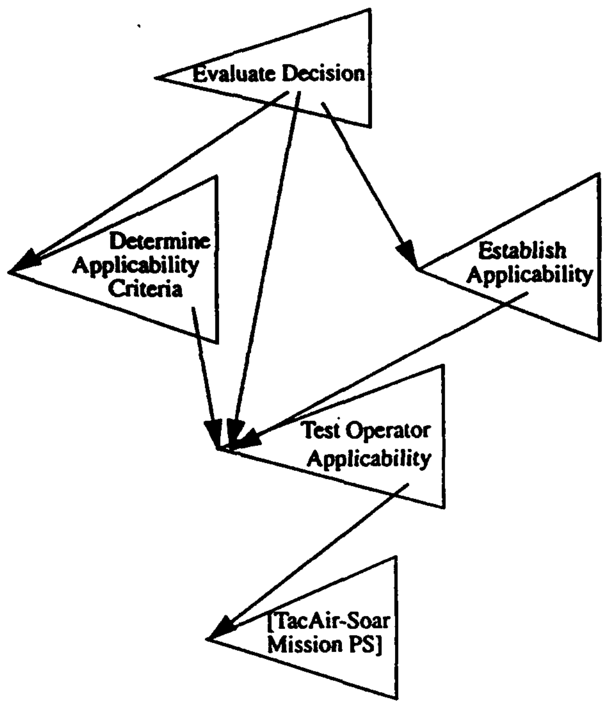


Figure 3: Problem spaces for evaluating decisions

called state. Then control passes to the Intercept problem space. If Intercept again selects the Employ-Weapons operator, Debrief knows that the recalled state description contains the information that motivated the original application of Employ-Weapons. Test Operator Applicability then immediately terminates the what-if simulation and returns to Evaluate-Decision an indication that the expected operator was in fact applied.

Reconsidering the original decision made during engagement is necessary for two reasons. First, it is possible that the recalled state does not correspond exactly to the situation in which the decision was originally made. This could happen if the TacAir-Soar operator was modifying working memory at the same time that working memory was being recorded. To deal with this case, a problem space called Establish-Applicability is employed to modify the recalled state by comparing it against the state associated with the immediately preceding event and trying to construct state intermediate between the two in which the operator can fire. But even if the right state was recalled it is important to reconsider the decision because it causes chunks to be built during the process. These chunks summarize the conditions in the recalled state that caused the Employ-Weapons operator to be selected. Implementation details internal to the Intercept problem space that caused the operator to be selected are automatically filtered out by the chunking process. The details of how this filtering occurs is beyond the scope of this paper, but please see [9].

Once it is determined that the recalled operator is applicable, the next step is to determine what would happen if the situation were slightly different from what was recalled. This helps to identify what the critical factors are in the state, and why they are critical. This analysis is performed in the Determine Applicability Criteria problem space. This space repeatedly deletes elements from the recalled state description and checks to see whether the originally selected operator would be applied. If the state change does not affect the applicability the the operator, a chunk previously built by Test-Operator-Applicability will fire, recognizing that the operator is still applicable. If the state change is significant, the chunk will not fire and what-if simulation will again be performed in the Test-Operator-Applicability space. If it is found that a different operator is selected, the name of the operator is returned to Determine-Applicability-Criteria, which then will perform further what-if analyses to determine why that operator was selected. Finally, the results of these analyses are returned as sets of significant attribute values associated with each selected operator. This is yet another point where Soar's learning mechanism is used to advantage. The next time TacAir-Soar applies the Employ-Weapons operator in a similar situation and Debrief is asked a question about it, it will immediately be able to recognize the similarity of the situation and produce an explanation, without having to perform any what-if simulation at all.

Explaining beliefs involves many of the same mechanisms that are used to evaluate decisions. Debrief searches backwards through the event history for the first event whose associated state includes the belief in question. Then Debrief removes the belief from the state description, and performs a what-if simulation of the operator associated with the event. If during simulation the belief is added back to the state, that indicates that the operator was responsible for asserting the belief into Soar's working memory. Determine-Applicability-Criteria can then be used to determine why that particular operator was selected.

8 Presenting Explanations

Once the necessary analysis is performed by Debrief, information is presented to the user. This presentation is performed via a hierarchical presentation planning process, initiated in the Present problem space shown in Figure 2. The planning process is similar to that of other multimedia generation systems [6, 1], although its ability to coordinate text and graphics is somewhat limited.

8.1 Selecting information to present

The first step in the presentation process is selecting what information should be presented. If the question that was asked involved evaluating a decision or belief, this step is trivial: every factor that was found to lead to the decision or belief in question is presented. If the user requested a summary of an event, however, the case is more complicated. Debrief has a wealth of information available about every event, in the form of the state information associated with the event and any substeps of the event. It must therefore determine which pieces of information are relevant.

Relevance is determined by constructing and maintaining a model of what the user is expected to know about the engagement. This is determined initially through a short questionnaire that the user fills out when he or she first sits down with the system. The user indicates the level of familiarity with the mission orders, and with what actually transpired during the engagement. Depending upon the answers to these questions Debrief will copy more or less information from TacAir-Soar's working memory into the user model. Later on when Debrief is planning a summary of a given event, it compares the recalled state associated with the event against the user model. If corresponding information is already in the user model then it is not presented.

If a piece of information is not present in the user model, Debrief next checks whether it is readily inferrable from other information that has already been selected for presentation. In particular, Debrief has knowledge about what facts are readily inferrable as consequences of particular events. Any such facts are omitted from the explanation.

After each event is described to the user, the user model is updated. All information that has been presented, or which is known to be inferrable from what was presented, is added to the model. However, it is assigned a lower degree of confidence—just because Debrief tells the user some fact does not mean that the user then knows it. If the user requests an elaboration about a particular event, information assigned to the user model with a low degree of confidence will still be presented.

8.2 Assigning information to media

Once information has been selected for presentation, Debrief must then determine what presentation media to use. It currently has knowledge of two presentation media: natural language and a graphical display of aircraft positions in 3-space. Each presentation medium is specified in terms of the types of information it is able to present: the graphical display is limited in its expressibility, whereas natural language is unlimited. Each piece of information is then allocated to the available media depending upon the type of presentation being given. If a summary description is being presented, only one medium will be selected, and graphical media will be preferred over textual media. Otherwise all available media will be used.

8.2.1 Generating the presentations

At the present time the only medium Debrief can employ is natural language, because the interface that will allow Debrief to control the display programmatically is not yet complete. As soon as the interface is complete, it will be possible for Debrief to start presenting the information that it is already able to assign to that medium. In the mean time, the presentations are in natural language instead. Natural language is produced using a simple sentence generator loosely based on Functional Unification Grammar [11].

9 Status and Evaluation

The Debrief system as it currently stands comprises thirteen problem spaces, implemented using eighty Soar operators and 1556 productions. It currently can describe and/or explain a total of 70 types of events. The natural language generation component has a vocabulary of 240 words and phrases. It has been used to describe and explain events occurring in 1 v 1 engagements, 1 v 2 engagements, and 2 v 1 engagements.

Formative evaluations of Debrief explanations have been performed with Navy Reserve fighter pilots. These evaluations confirmed that explanations are extremely helpful for validating the agent's performance, and building confidence in it. They also underscored the importance of having the agent justify its beliefs—the evaluators frequently wanted to ask questions about assertions made by Debrief during the course of the explanation. This experience motivated the work on incorporating the Explain-Belief question type into Debrief. Further evaluations and demonstrations are planned for later this year.

10 Acknowledgements

This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan, as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). The author wishes to thank the following people for their helpful comments and insights: Yolanda Gil, Paul Rosenbloom, William Swartout, Milind Tambe, and Lorna Zorman of USC / ISI; Bob Richards of BMH Inc.; Doug Holmes of ISX, Inc.; and Richard Lewis of Princeton University. Karl Schwamb contributed substantially to the development of the Debrief user interface.

References

[1] Y. Arens, E.H. Hovy, and M. Vossers. On the knowledge underlying multimedia presentations. In M. Maybury, editor, *Intelligent Multimedia Interfaces*. AAAI Press, 1993. to appear.

[2] W.J. Clancey. The advantages of abstract control knowledge in expert system design. In *Proceedings of the National Conference on Artificial Intelligence*, pages 74–78, Washington, DC, August 1983.

[3] W.J. Clancey. The epistemology of a rule-based expert system: A framework for explanation. *Artificial Intelligence*, 20(3):215–251, 1983.

[4] R. Davis. *Applications of Meta-Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases*. PhD thesis, Stanford University, 1976.

[5] R.B. Doorenbos. Matching 100,000 learned rules. In *Proceedings of the National Conference on Artificial Intelligence*, pages 290–296, Washington, DC, August 1993. AAAI.

[6] S.K. Feiner and K.R. McKeown. Coordinating text and graphics in explanation generation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 442–449, Anaheim, CA, August 1990. MIT Press.

[7] R.W. Hill and W.L. Johnson. Designing an intelligent tutoring system based on a reactive model of skill acquisition. In *Proceedings of the World Conference of Artificial Intelligence in Education*, pages 273–281, Edinburgh, Scotland, 1993.

[8] R.W. Hill and W.L. Johnson. Situated plan attribution for intelligent tutoring. In *Proceedings of the National Conference on Artificial Intelligence*, Seattle, Washington, 1994. to appear.

[9] W.L. Johnson. Agents that learn to explain themselves. In *Proceedings of the National Conference on Artificial Intelligence*, page forthcoming, Seattle, WA, August 1994. AAAI.

[10] W.G. Lehnert. *The Process of Question Answering*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1978.

[11] K.R. McKeown and M. Elhadad. *A Contrastive Evaluation of Functional Unification Grammar for Surface Language Generation: A Case Study in the Choice of Connectives*, pages 351–392. Kluwer Academic Publishers, Norwell, MA, 1991.

[12] J.D. Moore and W.R. Swartout. Pointing: A way toward explanation dialog. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 457–464, Anaheim, CA, August 1990. MIT Press.

[13] R. Neches, W.R. Swartout, and J.D. Moore. Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Transactions on Software Engineering*, SE-11(11):1337–1351, 1985.

[14] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.

[15] P.S. Rosenbloom, W.L. Johnson, R.M. Jones, F. Koss, J.E. Laird, J.F. Lehman, R. Rubinoff, K.B. Schwamb, and M. Tambe. Intelligent automated agents for tactical air simulation: A progress report. In *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, Orlando, FL, May 1994. Institute for Simulation and Training, University of Central Florida.

[16] W.R. Swartout and J.D. Moore. Explanation in second generation expert systems. In J.-M. David, J.-P. Krivine, and R. Simmons., editors, *Second Generation Expert Systems*. Springer-Verlag, 1993. To appear.

Biography

W. Lewis Johnson is a research assistant professor is a project leader at the University of Southern California Information Sciences Institute, and a research assistant professor in the USC Department of Computer Science. Dr. Johnson received his A.B. degree in Linguistics in 1978 from Princeton University, and his M.Phil. and Ph.D. degrees in Computer Science from Yale University in 1980 and 1985, respectively. He is interested in applying artificial intelligence techniques in the areas of computer-based training and software engineering. His current projects are developing tools that automate the generation of software documentation, and that explain the problem solving of intelligent agents.

Agents that Learn to Explain Themselves

W. Lewis Johnson
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
johnson@isi.edu

Abstract

Intelligent artificial agents need to be able to explain and justify their actions. They must therefore understand the rationales for their own actions. This paper describes a technique for acquiring this understanding, implemented in a multimedia explanation system. The system determines the motivation for a decision by recalling the situation in which the decision was made, and replaying the decision under variants of the original situation. Through experimentation the agent is able to discover what factors led to the decisions, and what alternatives might have been chosen had the situation been slightly different. The agent learns to recognize similar situations where the same decision would be made for the same reasons. This approach is implemented in an artificial fighter pilot that can explain the motivations for its actions, situation assessments, and beliefs.

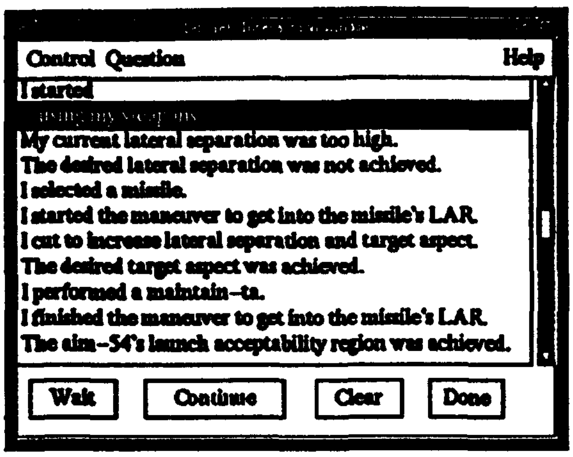
Introduction

Intelligent artificial agents need to be able to provide explanations and justifications for the actions that they take. This is especially true for computer-generated forces, i.e., computer agents that operate within battlefield simulations. Such simulations are expected to have an increasingly important role in the evaluation of missions, tactics, doctrines, and new weapons systems, and in training (Jones 1993). Validation of such forces is critical—they should behave as humans would in similar circumstances. Yet it is difficult to validate behavior through external observation; behavior depends upon the agent's assessment of the situation and its changing goals from moment to moment. Trainees can greatly benefit from automated forces that can explain their actions, so that the trainees can learn how experts behave in various situations. Potential users of computer-generated forces therefore attach great importance to explanation, just as potential users of computer-based medical consultation systems do (Teach & Shortliffe 1984).

Explanations based on traces of rule firings or paraphrases of rules tend not to be successful (Davis 1976; Swartout & Moore 1993; Clancey 1983b). They contain too many implementation details, and lack information about the domain and about rationales for the design of the system. More advanced explanation techniques encode domain knowledge and problem-solving strategies and employ them in problem solving either as metarules (Clancey 1983a) or in compiled form (Neches, Swartout, & Moore 1985). In the computer-generated forces domain, however, problem-solving strategies and domain knowledge representations are matters of current research. An intelligent agent in such a domain must integrate capabilities of perception, reactive problem solving, planning, plan recognition, learning, geometric reasoning and visualization, among others, all under severe real-time constraints. It is difficult to apply meta-level or compilation approaches in such a way that all of these requirements can be met at once.

This paper describes a system called Debrief that takes a different approach to explanation. Explanations are constructed after the fact by recalling the situation in which a decision was made, reconsidering the decision, and through experimentation determining what factors were critical for the decision. These factors are critical in the sense that if they were not present, the outcome of the decision process would have been different. Details of the agent's implementation, such as which individual rules applied in making the decision, are automatically filtered out. It is not necessary to maintain a complete trace of rule firings in order to produce explanations. The relationships between situational factors and decisions are learned so that they can be applied to similar decisions.

This approach of basing explanations on abstract associations between decisions and situational factors has similarities to the REX system (Wick & Thompson 1989). But while REX requires one to create a separate knowledge base to support explanation, Debrief automatically learns much of what it needs to know to generate explanations. The approach is related to techniques for acquiring domain models through experimentation (Gil 1993), except that the agent learns to model not the external world, but itself.



```
I started
   using my weapons
My current lateral separation was too high.
The desired lateral separation was not achieved.
I selected a missile.
I started the maneuver to get into the missile's LAR.
I cut to increase lateral separation and target aspect.
The desired target aspect was achieved.
I performed a maintain-ta.
I finished the maneuver to get into the missile's LAR.
The aim-54's launch acceptability region was achieved.
```

Figure 1: Part of a an event summary

Debrief is implemented as part of the TacAir-Soar fighter pilot simulation (Jones *et al.* 1993). Debrief can describe and justify decisions using a combination of natural language and diagrams. It is written in a domain-independent fashion so that it can be readily incorporated into other intelligent systems. Current plans call for incorporating it into the REACT system, an intelligent assistant for operators of NASA Deep Space Network ground tracking stations (Hill & Johnson 1994).

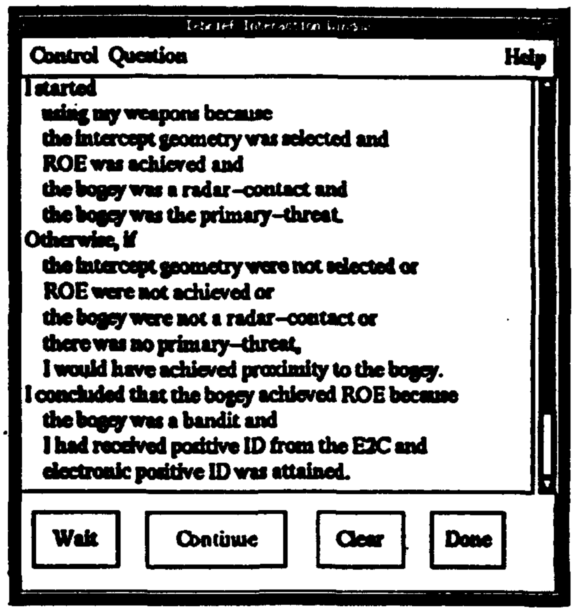
An Example

Consider the following scenario. A fighter is assigned a Combat Air Patrol (CAP) mission, i.e., it should fly a loop pattern, scanning for enemy aircraft. During the mission a bogey (an unknown aircraft) is spotted on the radar. The E2C, an aircraft whose purpose is to scan the airspace and provide information to the fighters, confirms that the bogey is hostile. The fighter closes in on the bogey, fires a missile which destroys the bogey, and then resumes its patrol.

After each mission it is customary to debrief the pilot. The pilot is asked to describe the engagement from his perspective, and explain key decisions along the way. The pilot must justify his assessments of the situation, e.g., why the bogey was considered a threat.

TacAir-Soar is able to simulate pilots executing missions such as this, and Debrief is able to answer questions about them. TacAir-Soar controls a simulation environment called ModSAF (Calder *et al.* 1993) that simulates the behavior of military platforms. TacAir-Soar receives information from ModSAF about aircraft status and radar information, and issues commands to fly the simulated aircraft and employ weapons. After an engagement users can interact with Debrief to ask questions about the engagement.

The following is a typical interaction with Debrief.



```
I started
   using my weapons because
   the intercept geometry was selected and
   ROE was achieved and
   the bogey was a radar-contact and
   the bogey was the primary-threat.
Otherwise, if
   the intercept geometry were not selected or
   ROE were not achieved or
   the bogey were not a radar-contact or
   there was no primary-threat,
   I would have achieved proximity to the bogey.
I concluded that the bogey achieved ROE because
   the bogey was a bandit and
   I had received positive ID from the E2C and
   electronic positive ID was attained.
```

Figure 2: Explanations of the agent's decisions

Questions are entered through a window interface, by selecting a type of question and pointing to the event or assertion that the question refers to. The first question selected is of type Describe-Event, i.e., describe some event that took place during the engagement; the event chosen is the entire mission. Debrief then generates a summary of what took place during the mission. The user is free to select statements in the summary and ask follow-on questions about them.

Figure 1 shows part of a typical mission summary. One of the statements in the summary, "I started using my weapons," has been selected by the user, so that a follow-on question may be asked about it. Figure 2 shows the display at a later point in the dialog, after follow-on questions have been asked. First, a question of type Explain-Action was asked of the decision to employ weapons, i.e., explain why the agent chose to perform this action. The explanation appears in the figure, beginning with the sentence "I started using my weapons because the intercept geometry was selected and..." Debrief also lists an action that it did not take, but might have taken under slightly different circumstances: flying toward the bogey to decrease distance.

One can see that the agent's actions are motivated largely by previous assessments and decisions. The bottom of Figure 2 shows the answer to a follow-on question relating to one of those assessments, namely "ROE was achieved,"[1] Debrief lists the following fac-

[1]ROE stands for Rules of Engagement, i.e., the conditions under which the fighter is authorized to engage the enemy.

tors: the bogey was known to be hostile (i.e., a "bandit"), the bogey was identified through electronic means and confirmation of the identification was obtained from the E2C.

In order to answer such questions, Debrief does the following. First, it recalls the events in question and the situations in which the events took place. When summarizing events, it selects information about the intermediate states and subevents that should be presented, selects appropriate media for presentation of this information (the graphical display and/or natural language), and then generates the presentations. To determine what factors in the situation led to the action or conclusion, Debrief invokes the TacAir-Soar problem solver in the recalled situation, and observes what actions the problem solver takes. The situation is then repeatedly and systematically modified, and the effects on the problem solver's decisions are observed. Beliefs are explained by recalling the situation in which the beliefs arose, determining what decisions caused the beliefs to be asserted, and determining what factors were responsible for the decisions.

Implementation Concerns

Debrief is implemented in Soar, a problem-solving architecture that implements a theory of human cognition(Newell 1990). Problems in Soar are represented as goals, and are solved within problem spaces. Each problem space consists of a state, represented as a set of attribute-value pairs, and a set of operators. All processing in Soar, including applying operators, proposing problem spaces, and constructing states, is performed by productions. During problem solving Soar repeatedly selects and applies operators to the state. When Soar is unable to make progress, it creates a new subgoal and problem space to determine how to proceed. Results from these subspaces are saved by Soar's chunking mechanism as new productions, which can be applied to similar situations.

The explanation techniques employed in Debrief are not Soar-specific; however, they do take advantage of certain features of Soar.

- The explicit problem space representation enables Debrief to monitor problem solving when constructing explanations.
- Since Soar applications are implemented in production rules, it is fairly straightforward to add new rules for explanation-related processing.
- Learning enables Debrief to reuse the results of previous explanation processing, and build up knowledge about the application domain.

The current implementation of Debrief consists of thirteen Soar problem spaces. Two are responsible for inputing questions from the user, three recall events and states from memory, four determine the motivations for actions and beliefs, three generate presentations, and one provides top-level control. The following sections describe the system components involved in determining motivations for decisions and beliefs; other parts of the system are described in (Johnson 1994).

Memory and Recall

In order for Debrief to describe and explain decisions, it must be able to recall the decisions and the situations in which they occurred. In order words, the agent requires an episodic memory. Debrief includes productions and operators that execute during the problem solving process in order to record episodic information, and a problem space called Recall-State that reconstructs states using this episodic information.

The choice of what episodic information to record is determined by a specification of the agent's working memory state. This specification identifies the state attributes that are relevant for explanation, and identifies their properties, e.g., their cardinality and signature, and how the attribute values may change during problem solving. In order to apply Debrief to a new problem solver, it is necessary to supply such a specification for the contents of the problem solver's working memory, and indicate which operators implement decisions what should be explainable. However, it is not necessary to specify how the problem solver uses its working memory in making decisions—that is determined by Debrief automatically.

When the problem solver applies an operator that as marked as explainable, Debrief records the operator application in a list of events that took place during the problem solving. It also records all attribute values that have changed since the last problem solving event that was recorded.

Debrief then builds chunks that associate the state changes with the problem solving event. Once these chunks are built, the state changes can be deleted from working memory, because the chunks are sufficient to enable Debrief to recall the working memory state. During explanation, when Debrief needs to recall the state in which a problem solving event occurred, it invokes the Recall-State problem space. This space reconstructs the state by proposing possible attribute values; the chunks built previously fire, selecting the value that was associated with the event. Recall-State aggregates these values into a copy of the state at the time of the original event, and returns it. This result is chunked as well, enabling Debrief immediately to recall the state associated with the event should it need to refer back to it in the future. This process is an instance of data chunking, a common mechanism for knowledge-level learning in Soar systems (Rosenbloom, Laird, & Newell 1987).

Debrief thus makes extensive use of Soar's long term memory, i.e., chunks, in constructing its episodic memory. In a typical TacAir-Soar run several hundred such chunks are created. This is more economical than simply recording a trace of production firings, since over

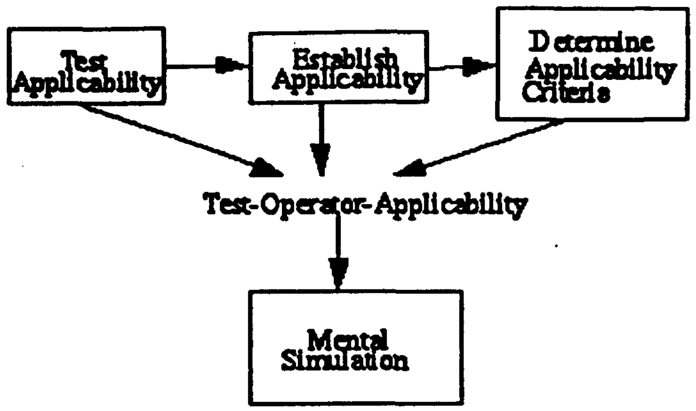


Figure 3: The process of evaluating decisions

6000 productions fire in a typical TacAir-Soar run. Since Soar has been shown be able to handle memories containing hundreds of thousands of chunks (Doorenbos 1993), there should be little difficulty in scaling up to more complex problem solving applications.

Explaining Actions and Conclusions

Suppose that the user requests the motivation for the action "I started using my weapons." Debrief recalls the type of event involved, operator that was applied, the problem space in which it was applied, and the problem solving state. In this case the event type is Start-Event, i.e., the beginning of an operator application, the operator is named Employ-Weapons, and the problem space is named Intercept. The situation was one where the agent had decided to intercept the bogey, and had just decided what path to follow in performing the intercept (called the intercept geometry).

Analysis of recalled events such as this proceeds as shown if Figure 3. The first step, testing applicability, verifies that TacAir-Soar would select an Employ-Weapons operator in the recalled state. An operator called Test-Operator-Applicability performs the verification, by setting up a "mental simulation" of the original decision, and monitoring it to see what operators are selected.

This initial test of operator applicability is important for the following reasons. State changes are not recorded in episodic memory until the operator has already been selected. The operator might therefore modify the state before Debrief has a chance to save it, making the operator inapplicable. This is not a problem in the case of Employ-Weapons, but if it were Debrief would attempt to establish applicability, which involves recalling the state immediately preceding the state of the event, and trying to find an interpolation of the two states in which the operator would be selected. But even when recalling the precise problem solving state is not a problem, verifying applicability is useful because it causes chunks to be built that facilitate subsequent analysis.

After a state has been found in which the recalled operator is applicable, the next step is to determine applicability criteria, i.e., identify what attributes of the state are responsible for the operator being selected. This also involves applying the Test-Operator-Applicability operator to construct mental simulations.

Mental simulation

Given the problem space Intercept, the recalled state, the operator Employ-Weapons, and the decision Start-event(Employ-Weapons), Test-Operator-Applicability operates as follows. It creates an instance of the Intercept problem space as a subspace, and assigns as its state a copy of the recalled state. The working memory specification described above is helpful here: it determines which attributes have to be copied. This state is marked as a simulation state, which activates a set of productions responsible for monitoring mental simulations. Test-Operator-Applicability copies into the simulation state the event and the category of decision being evaluated. There are three such categories: perceptions, which recognize and register some external stimulus, conclusions, which reason about the situation and draw inferences from it, and actions, which are operations that have some effect on the external world. Employ-Weapons is thus an action. The Intercept problem space is disconnected from external sensors and effectors (the ModSAF simulator), so that mental simulation can be freely performed. Execution then begins in the problem space. The first operator that is selected is Employ-Weapons. The monitoring productions recognize this as the desired operator, return a flag to the parent state indicating that the desired event was observed, and the mental simulation is terminated. If a different operator or event had been selected instead, Debrief would be checked to see if it is of the same category as the expected operator, i.e., another action. If not, simulation is permitted to continue; otherwise simulation is terminated and the a description of the operator that applied instead is returned.

Whenever a result is returned from mental simulation, a chunk is created. Such chunks may then be applicable to other situations, making further mental simulation unnecessary. Figure 4 shows the chunk that is formed when Debrief simulates the selection of the Employ-Weapons operator. The conditions of the chunk appear before the symbol $\rightarrow$ and actions follow. Variables are symbols surrounded by angle brackets, and attributes are preceded by a carat ($\wedge$). The conditions include the expected operator, Employ-Weapons, the problem space, Intercept, and properties of the state, all properties of the bogey. If the operator is found to be inapplicable, a different chunk is produced, that indicates which operator is selected instead of the expected one.

```
(sp chunk-230 :chunk
  (goal <g1> ^operator <o1> ^state <s1>)
  (<o1> ^name test-operator-applicability
        ^expected-operator employ-weapons
        ^expected-step *none*
        ^problem-space intercept)
  (<s1> ^simulated-state <r1>)
  (<r1> ^local-state <l1>)
  (<l1> ^bogey <b1>)
  (<b1> ^intention known-hostile
        ^roe-achieved *yes*
        ^intercept-geometry-selected *yes*
        ^contact *yes*)
  (<l1> ^primary-threat <b1>)
-->
  (<s1> ^applicable-operator employ-weapons))
```

Figure 4: An example chunk

These chunks built during mental simulation have an important feature—they omit the details of how the operator and problem space involved is implemented. This is an inherent feature of the chunking process, which traces the results of problem solving in a problem space back to elements of the supergoal problem space state. In this case the state recalled from episodic memory is the part of the supergoal problem space state, so elements of the recalled state go into the left hand side of the chunk.

Determining the cause for decisions

At this point it would be useful to examine the chunks built during mental simulation in order to proceed to generate the explanation. Unfortunately, productions in a Soar system are not inspectable within Soar. This limitation in the Soar architecture is deliberate, reflecting the difficulty that humans have in introspecting on their own memory processes. It does not a serious problem for Debrief, because the chunks built during mental simulation can be used to recognize which attributes of the state are significant.

The identification of significant attributes is performed in the Determine-Applicability-Criteria problem space, which removes attributes one by one and repeatedly applies Test-Operator-Applicability. If a different operator is selected, then the removed attribute must be significant. If the value of a significant attribute is a complex object, then each attribute of that object is analyzed in the same way; the same is true for any significant values of those attributes. Meanwhile, if the variants resulted in different operators being selected, the applicability criteria for these operators are identified in the same manner. This generate-and-test approach has been used in other Soar systems to enlist recognition chunks in service of problem solving (Vera, Lewis, & Lerch 1993), and is similar to Debrief's mechanism for reconstructing states from episodic memory. Since the state representations are hierarchically organized, the significant attributes are found quickly.

If chunking were not taking place, Debrief would be performing a long series of mental simulations, most of which would not yield much useful information. But the chunks that are created help to ensure that virtually every mental simulation uncovers a significant attribute, for the following reason. Subgoals are created in Soar only when impasses occur. Test-Operator-Applicability instantiates the mental simulation problem space because it tries to determine whether the recalled operator is applicable, is unable to do so, and reaches an impasse. When chunks such as the one in Figure 4 fire, they assert that the operator is applicable, so no impasse occurs. Mental simulation thus occurs only in situations that fail to match the chunks that have been built so far. In the case of the Employ-Weapons operator, a total of seven mental simulations of variant states are required: two to determine that the bogey is relevant, and five to identify the bogey's relevant attributes.

Furthermore, even these mental simulations become unnecessary as Debrief gains experience explaining missions. Suppose that Debrief is asked to explain a different Employ-Weapons event. Since most of the significant features in the situation of this new event are likely to be similar to the significant features of the previous situation, the chunks built from the previous mental simulations will fire. Mental simulation is required for the situational features that are different, or if the operator was selected for different reasons.

Two kinds of chunks are built when Determine-Applicability-Criteria returns its results. One type identifies all of the significant features in the situation in which the decision was made. The other type identifies an operator that might have applied instead of the expected operator, and the state in which the operator applies. These chunks are created when mental simulation determines that an operator other than the expected one is selected. Importantly, the chunks fire whenever a similar decision is made in a similar situation. By accumulating these chunks Debrief thus builds an abstract model of the application domain, associating decisions with their rationales and alternatives. The problem solver's performance-oriented knowledge is reorganized into a form suited to supporting explanation.

Performing mental simulation in modified states complicates mental simulation in various respects. The result of deleting an attribute is often the selection of an operator in mental simulation to reassert the same attribute. Debrief must therefore monitor the simulation and detect when deleted attributes are being reasserted. The modified state may cause the problem solver to fail, resulting in an impasse. Mental simulation must therefore distinguish impasses that are a normal result of problem solving from impasses that suggest that the problem solver is in an erroneous state.

There is one shortcoming of the analysis technique described here. Chunking in Soar cannot always backtrace through negated conditions in the left hand sides of productions. Therefore if the problem solver opted for a decision because some condition was *absent* in the situation, Debrief may not be able to detect it. Developers of Soar systems get around this problem in chunking by using explicit values such as *unknown* to indicate that information is absent. This same technique enables Debrief to identify the factors involved.

Relationship to other exploratory learning approaches

The closest correlate to Debrief's decision evaluation capability is Gil's work on learning by experimentation (Gil 1993). Gil's EXPO system keeps track of operator applications, and the states in which those operators were applied. If an operator is found to have different effects in different situations, EXPO compares the states to determine the differences. Another system by Scott and Markovich (Scott & Markovich 1993) performs an operation on instances of a class of objects, to determine whether it has different effects on different members of the class. This enables it to discover discriminating characteristics within the class.

Some exploratory learning systems, such as Rajamoney's systems (Rajamoney 1993), invest significant effort to design experiments that provide the maximum amount of information. This is necessary because experiments can be costly and can have persistent effects on the environment. Debrief's chunking-based technique filters out irrelevant experiments automatically, without significant effort. Side events on the environment are not a concern during mental simulation.

Explaining Beliefs

Explaining beliefs, e.g., that ROE was achieved, involves many of the same analysis steps used for explaining decisions. Debrief starts by searching memory for the nearest preceding state in which the belief came to be held. It determines what operator was being applied during that state, and uses Establish-Applicability if necessary to make sure that the operator applies in the recalled state. If the belief had to be retracted in order to make Test-Operator-Applicability succeed, then the operator was responsible for asserting the belief. Such is the case for the belief that ROE is achieved, which is asserted by an operator named ROE-Achieved. Otherwise, Debrief would remove the belief and attempt mental simulation again; if the belief is asserted in the course of applying the operator, the operator is probably responsible for the belief.

Summary of the Effects of Learning

Learning via chunking takes place throughout the Debrief system. The following is a summary of the different types of chunks that are produced:

- Episodic memory recognition chunks: event + attribute value → recognition;
- State recall chunks: event → state;
- Mental simulation chunks: event + problem space + state → applicable or inapplicable + alternative operator;
- Applicability analysis chunks: event + problem space + state → significant state attributes; event + problem space + state → alternative operator + alternative state;
- Natural language generation chunks: case frame → list of words; content description → list of utterances;
- Presentation chunks: content description + user model → utterances + media control commands + user model updates.

The presentation mechanisms that yield the latter two types of chunks are described in (Johnson 1994). Altogether, these chunks enable Debrief to acquire significant facility in explaining problem solving behavior. These chunks result in speedups during the course of explaining a single mission. Future experiments will determine the transfer effects between missions.

Evaluation and Status

The implementation of Debrief comprises over 1700 productions; in a typical session these are augmented by between 500 and 1000 chunks. Debrief currently can describe and/or explain a total of 66 types of events in the tactical air domain. Its natural language generation component has a vocabulary of 259 words and phrases. Debrief can explain a range of one-on-one and one-on-two air-to-air engagements.

Formative evaluations of Debrief explanations have been performed with US Naval Reserve fighter pilots. These evaluations confirmed that explanations are extremely helpful for validating the agent's performance, and building confidence in it. They also underscored the importance of having the agent justify its beliefs—the evaluators frequently wanted to ask questions about assertions made by Debrief during the course of the explanation. This motivated the development of support for the Explain-Belief question type. There was immediate interest on the part of the subject matter experts in using Debrief to understand and validate the behavior of TacAir-Soar agents.

The weakest point of the current system is its natural language generation capability. However, this was found not to be a major concern for the evaluators. Their primary interest was in understanding the thinking processes of TacAir-Soar, and to the extent that Debrief made that reasoning apparent it was considered effective.

Conclusion

This paper has described a domain-independent technique for analyzing the reasoning processes of an intelligent agent in order to support explanation. This technique reduces the need for extensive knowledge acquisition and special architectures in support of explanation. Instead, the agent can construct explanations on its own. Learning plays a crucial role in this process. Next steps include extending the range to questions that can be answered, improving the natural language generation, and making greater use of multi-media presentations. There is interest in using the mental simulation framework described here to improve the agent's problem solving performance, by discovering alternative decision choices with improved outcomes.

Acknowledgements

The author wishes to thank Paul Rosenbloom, Milind Tambe, and Yolanda Gil for their helpful comments. Dr. Johnson was supported in part by the ARPA and the Naval Research Laboratory under contract number N00014-92-K-2015 (via a subcontract from the University of Michigan). Views and conclusions contained in this paper are the author's and should not be interpreted as representing the official opinion or policy of the U.S. Government or any agency thereof.

References

Calder, R.; Smith, J.; Courtemanche, A.; Mar, J.; and Ceranowicz, A. 1993. ModSAF behavior simulation and control. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, 347–359. Orlando, FL: Institute for Simulation and Training, University of Central Florida.

Clancey, W. 1983a. The advantages of abstract control knowledge in expert system design. In *Proceedings of the National Conference on Artificial Intelligence*, 74–78.

Clancey, W. 1983b. The epistemology of a rule-based expert system: A framework for explanation. *Artificial Intelligence* 20(3):215–251.

Davis, R. 1976. *Applications of Meta-Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases*. Ph.D. Dissertation, Stanford University.

Doorenbos, R. 1993. Matching 100,000 learned rules. In *Proceedings of the National Conference on Artificial Intelligence*, 290–296. Menlo Park, CA: AAAI.

Gil, Y. 1993. Efficient domain-independent experimentation. Technical Report ISI/RR-93-337, USC / Information Sciences Institute. Appears in the Proceedings of the Tenth International Conference on Machine Learning.

Hill, R., and Johnson, W. 1994. Situated plan attribution for intelligent tutoring. In *Proceedings of the National Conference on Artificial Intelligence*.

Johnson, W. 1994. Agents that explain their own actions. In *Proc. of the Fourth Conference on Computer Generated Forces and Behavioral Representation*. Orlando, FL: Institute for Simulation and Training, University of Central Florida. World Wide Web access: http://www.isi.edu/soar/debriefable.html.

Jones, R.; Tambe, M.; Laird, J.; and Rosenbloom, P. 1993. Intelligent automated agents for flight training simulators. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, 33–42. Orlando, FL: Institute for Simulation and Training, University of Central Florida.

Jones, R. 1993. Using CGF for analysis and combat development. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, 209–219. Orlando, FL: Institute for Simulation and Training, University of Central Florida.

Neches, R.; Swartout, W.; and Moore, J. 1985. Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Transactions on Software Engineering* SE-11(11):1337–1351.

Newell, A. 1990. *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.

Rajamoney, S. 1993. The design of discrimination experiments. *Machine Learning* 185–203.

Rosenbloom, P.; Laird, J.; and Newell, A. 1987. Knowledge level learning in Soar. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 618–623. Menlo Park, CA: American Association for Artificial Intelligence.

Scott, P., and Markovich, S. 1993. Experience selection and problem choice in an exploratory learning system. *Machine Learning* 49–68.

Swartout, W., and Moore, J. 1993. Explanation in second generation expert systems. In David, J.-M.; Krivine, J.-P.; and Simmons., R., eds., *Second Generation Expert Systems*. Springer-Verlag. 543–585.

Teach, R., and Shortliffe, E. 1984. An analysis of physicians' attitudes. In Buchanan, B., and Shortliffe, E., eds., *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley. 635–652.

Vera, A.; Lewis, R.; and Lerch, F. 1993. Situated decision-making and recognition-based learning: Applying symbolic theories to interactive tasks. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, 84–95. Hillsdale, NJ: Lawrence Erlbaum Associates.

Wick, M., and Thompson, W. 1989. Reconstructive explanation: Explanation as complex problem solving. In *Proceedings of the Eleventh Intl. Joint Conf. on Artificial Intelligence*, 135–140. San Mateo, CA: Morgan Kaufmann.

Multiple Information Sources and Multiple Participants: Managing Situational Awareness in an Autonomous Agent

Randolph M. Jones and John E. Laird
Artificial Intelligence Laboratory
University of Michigan
1101 Beal Avenue
Ann Arbor, MI 48109-2110

Abstract

One of the most important tasks in a tactical engagement is to maintain awareness of the current situation. This is as true for simulated intelligent agents as it is for humans in real engagements. We have identified two key capabilities that are required for maintaining situational awareness: managing and synthesizing information from a variety of information sources, and correctly identifying and sorting engagement participants into an appropriate mental representation. This paper discusses our efforts in addressing these capabilities within the TacAir-Soar system.

An intelligent simulated agent must be able to observe and interpret the world it is operating in. This includes observing how the world reacts to behaviors generated by the agent, as well as behaviors generated by other agents within the simulation. As the world changes, the agent must continuously build and maintain a "mental picture" of the world's current state (i.e., maintain *situational awareness*). Otherwise there is no hope of generating appropriate behaviors to accomplish the agent's goals. In order to maintain such a picture, the agent should use whatever information sources it has available. In general, more information sources are better, but having multiple sources demands that the agent be able to synthesize the different types of information in order to form a representation of the world that is as complete and correct as possible.

In earlier work (Jones, Tambe, Laird, & Rosenbloom, 1993) we concentrated on building agents that generated reasonable behavior given rather strong assumptions about world information. The past TacAir-Soar agent assumed that there were at most two agents operating in the simulated tactical air environment: the agent itself and one potential enemy. In addition, the agent had only two sources of information: cockpit controls reported information about the agent's vehicle and weapons, and a radar reported information about the other participant in the environment.

This arrangement allowed the system to generate some tactical behaviors, but it greatly limited the types of situations in which TacAir-Soar could function. More typically, a tactical air agent finds itself in situations similar to that shown in Figure 1. There can be a number of participants in the engagement, and a number of ways to gather information about these agents.

Thus, in our current work, we have expanded the abilities of the TacAir-Soar agent to manage information. The current agent is able to maintain mental representations of any number of other participants in the simulation.[1] In addition, the agent now synthesizes information from a number of different sources. Each agent receives information visually, from its radar, and via radio from other participants in the engagement. These increases in capabilities are necessary in order for TacAir-Soar to function reasonably in the complex domain of tactical flight. However, they also introduce a number of complexities to the task of maintaining situational awareness, or keeping a mental picture of what is happening in the world. We feel that maintaining situational awareness boils down to two cognitive capabilities: managing information from multiple sources and managing information about multiple participants in an engagement. This paper discusses our approach to addressing these two broad issues within the TacAir-Soar system.

Managing multiple information sources

Besides receiving information from its own vehicle's instruments and gauges, the current version of TacAir-Soar receives information about other participants from three basic sources. The agent may receive information from a visual contact with another simulation participant such as another airplane (this information comes in through a DIS *visual object* package). The agent may achieve a radar contact with a participant. Finally, the agent may receive communicated information about another participant (this information may come from a ground controller, an air controller, or perhaps a section or division partner). In addition, TacAir-Soar periodically records position information for current contacts. Thus, when no *active* (visual, radar, or communication) contact information is available, the agent's memory becomes a fourth source of information.

When there is only one active information source, things are relatively simple. The system simply uses the information available to track the contact. This may not always be the best or most up-to-date information, but the system can only make do with what it has. When there are multiple active information sources describing contact with a particular participant, difficulties may arise. In this case, there is a decision to be made about where to look for the correct information. Some types of information are only available from particular types of sources, but others are provided by all of the information sources. For example, both radar and visual information can provide the relative position of another airplane, but radar can provide a more accurate measurement of the airplane's altitude and speed. In addition, information sources have different update rates, so some may contain "stale" information at certain

[1] In theory this number is unbounded, but in practice agent performance can degrade dramatically when it has too many other agents to pay attention to.

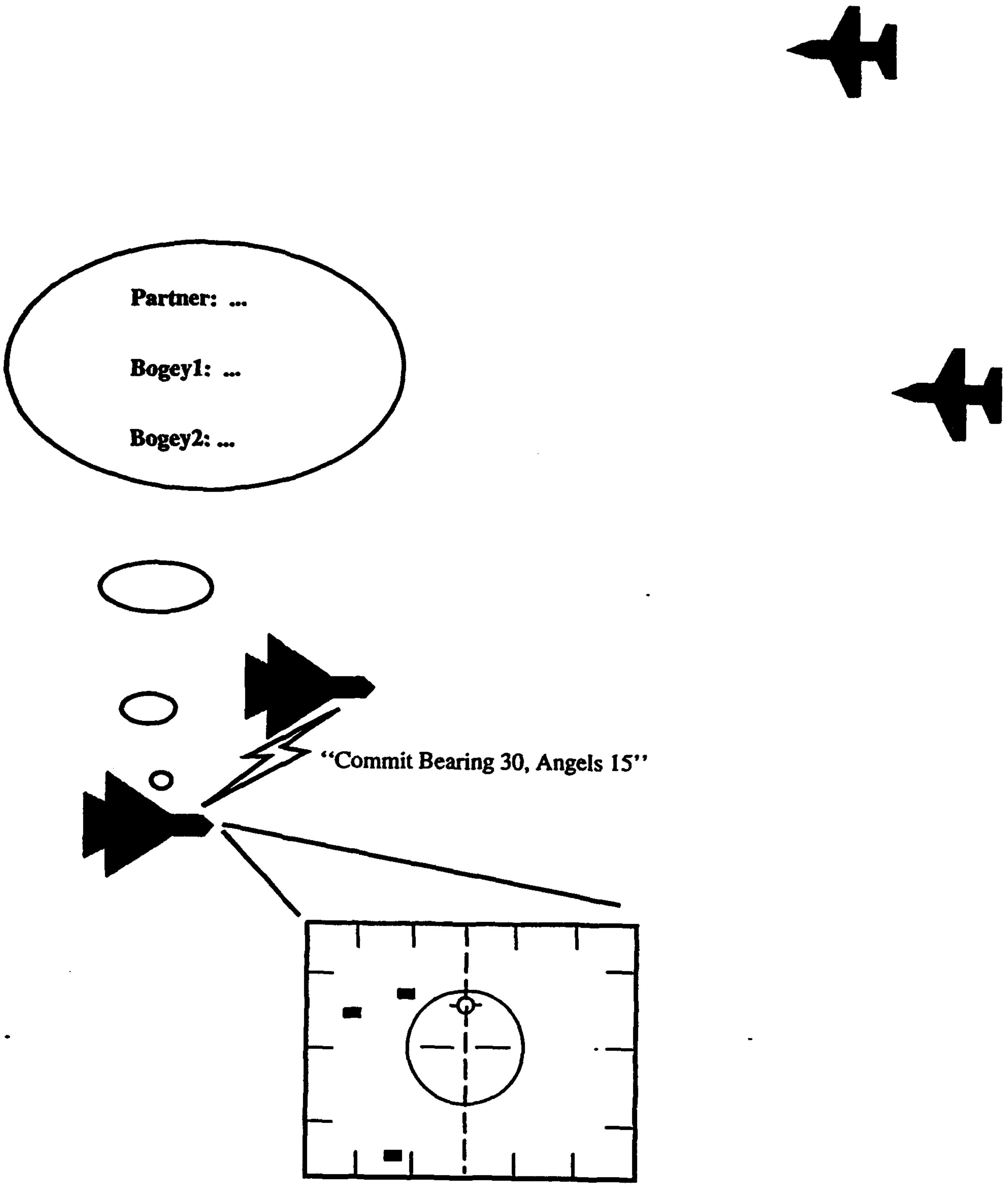


Figure 1. A typical engagement involving multiple participants and multiple sources of information.

times. For example, visual information has an almost instantaneous update rate, radar information depends on the speed of the actual radar sensors, and communicated information is updated relatively slowly. TacAir-Soar currently prioritizes its information sources by assuming that visual information is generally better than radar, radar is generally better than communication, and communication is generally better than memorized information. In addition, the system ranks current information by remembering how long it has been since the information was last updated. When it wants to look up information for a particular participant, it uses information from the best existing source.

Another issue involves what actions TacAir-Soar should take in order to gain new information about a participant. In the older version of the system this was a simple matter because there was only a radar information source. If the system did not have radar contact, it did what it could to achieve a radar contact. Now, however, there are many different information sources and different ways to achieve them. In addition, some sources are better than others in different situations. For example, radar is good at accurately tracking altitudes and headings at a distance, but rapid visual information is necessary as the engagement progresses.

For example, if TacAir-Soar only has recorded information about an agent, the system might request some communicated information in order to achieve a better idea of what the agent is doing. Communicated information is useful (particularly at long ranges), but it is somewhat inaccurate and takes a while to report. Thus, the system still does what it can to get a radar or visual contact in order to get faster, more reliable information. The point here is that TacAir-Soar not only requires knowledge for managing information from multiple sources, but it also must have the knowledge to *seek out* different types of information contacts when appropriate.

Identifying and tracking multiple participants

In the tactical air domain, there are generally a number of participants in each engagement. A particular simulated agent will probably have a section partner, and there may be any number of other friendly and hostile participants that the agent must worry about. The major difficulty arises in creating a mapping between the participants that "are really out there" and the participants that the agent is currently receiving information about (from at least one of the information sources). Thus, much of our research effort has been on finding an efficient, accurate, and realistic method to maintain this mapping.

The problem can be summarized as follows. The agent may have a mental representation of a number of other participants in the current simulation (we will refer to these as *mental agents*). Now the agent receives a new *contact* (i.e., new visual, radar, or communicated information becomes available). The agent must now decide whether this new contact corresponds to one of the mental agents, or whether this is a new participant (requiring the creation of a new mental agent). If the contact corresponds to one of the existing mental agents, TacAir-Soar must decide *which* mental agent the information pertains to. Only after this mapping has been completed can the system correctly interpret and respond to the new information. This should be done as quickly as possible, but it should also be done with the same intelligence and flexibility that human pilots have. It can be disastrous, for example, to conclude by mistake that a hostile participant is the agent's section partner.

A similar problem arises in the case where two agents must communicate with each other about other participants in the engagement. For example, the lead agent of a flight section may need to tell its partner which bogey it is targeting. However, the two agents will not necessarily have the same mental agents represented, and often they will not even have the same information coming in on their sensors. The solution for this is for the lead to describe particular characteristics of the bogey, so the partner can use this information to determine which mental agent the lead is talking about. In TacAir-Soar, the problem of communicating about other engagement participants is subsumed by the general problem of identifying and sorting incoming information (regardless of the particular information source) to the appropriate mental agent.

TacAir-Soar solves this problem by passing new information through a set of filters. The first filter determines whether the new information closely matches any existing contact information for a mental agent (e.g., the system might achieve radar contact with an agent for which it had only previously received communicated information). If this filter fails to identify a unique mental agent, the next filter compares any new position information to the last position information the system recorded for each mental agent. TacAir-Soar uses a form of temporal reasoning, based on the time of the last recorded position for each mental agent, together with the contact's heading, speed, etc., to determine which mental agents the new contact information could *possibly* pertain to.

This filter may rule out any existing mental agents, in which case TacAir-Soar will create a new one. On the other hand, the filter may provide a unique mental agent to assign the new contact information to. Otherwise, there is still some ambiguity so the system must use its final filter. This filter compares individual features in the new contact information to the same features in each remaining candidate mental agent. The mental agents that are closest in value for a chosen feature are saved, while others are eliminated from consideration. This process continues through a set of features until a unique mental agent remains. Currently, the features that TacAir-Soar examines are magnetic bearing, range, altitude, speed, and heading. If, after sorting through all of these features, there is still more than one candidate mental agent, the system simply chooses one at random. However, this is rare unless two contacts appear in almost the same position, in which case further discrimination is probably meaningless anyway.

This filtering mechanism is based on the methods that real Navy pilots and RIOs use to identify contacts, and it has proved relatively robust in allowing the

TacAir-Soar agent to reason about multiple participants in a simulated engagement. However, there are times when the current mechanism fails, indicating that there is some knowledge missing from the process. For example, human pilots generally begin a mission with an idea of where the friendly and enemy forces are, and this helps them identify initial contacts. Additional information sources, such as IFF, can also be used to help identify and sort contacts. Within visual range, pilots can use the actual shapes of different vehicle types to determine who is wh[illegible]o far TacAir-Soar does not use these a[illegible]tional types of knowledge, and so it is prone to getting confused in some situations where humans do not have difficulties maintaining situational awareness. TacAir-Soar can also become confused when engagements become fast and close, so it does not have time to sort and process all of the incoming information properly. However, this is the type of situation that is difficult even for human experts.

Summary

Maintaining situational awareness is a particularly important part of tactical behavior, and simulated tactical agents must address the issues involved. We have identified two important components of maintaining situational awareness: managing knowledge about multiple tactical participants in an engagement, and managing incoming information from a variety of sources. In addition, we have implemented knowledge and behaviors that address these issues into the TacAir-Soar system.

In order to reason about multiple information sources, the system has mechanisms for choosing between existing sources, as well as methods for generating behavior so that the agent can acquire new information (such as searching for radar contacts or moving into visual range). Reasoning about multiple participants requires the agent to form a mental picture of its situation, including a mental representation of each participant in the engagement. As new information is acquired, the system uses heuristics to determine to which mental agent each new contact pertains. In addition, the agent performs these tasks within the dynamic constraints of the domain, so it is possible for it to get confused in the same types of situations as humans.

Our continuing work will focus on the addition of new information sources such as IFF and radar-warning receivers. Together with these devices, the agent will require the knowledge to gather and manage the types of information these devices provide in the appropriate situations. In addition, we are continuing to study how human pilots maintain knowledge about other participants in an engagement, so that we can improve the mechanisms for identifying and sorting contacts into mental agent representations. As this knowledge improves, we expect to develop general intelligent methods for maintaining situational awareness, so the agent can generate even more realistic and appropriate behavior.

Acknowledgements

This research has benefited from the authors' interactions with all of the members of the Soar-IFOR group at the University of Michigan, University of Southern California, and Carnegie Mellon University. The members of BMH, Inc. have proved invaluable as subject-matter experts. The research is supported by contract N00014-02-K-2015 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Research Laboratory.

References

Jones, R. M., Tambe, M., Laird, J. E., & Rosenbloom, P. S. (1993). Intelligent automated agents for flight training simulators. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation* (pp. 33–42). Orlando, FL.

Biographies

Randolph M. Jones received his Ph.D. in Information and Computer Science from the University of California, Irvine, in 1989. He is currently an assistant research scientist in the Artificial Intelligence Laboratory at the University of Michigan. His research interests lie in the areas of intelligent agents, problem solving, machine learning, and psychological modeling.

John E. Laird is an associate professor of Electrical Engineering and Computer Science and the director of the Artificial Intelligence Laboratory at the University of Michigan. He received his B.S. degree in Computer and Communication Sciences from the University of Michigan in 1975 and his M.S. and Ph.D. degrees in Computer Science from Carnegie Mellon University in 1978 and 1983, respectively. His interests are centered on creating integrated intelligent agents (using the Soar architecture), leading to research in problem solving, complex behavior representation, machine learning, and cognitive modeling.

Generating Behavior in Response to Interacting Goals

Randolph M. Jones,[1] John E. Laird,[1] Milind Tambe,[2] and Paul S. Rosenbloom[2]

[1]Artificial Intelligence Laboratory
University of Michigan
1101 Beal Avenue
Ann Arbor, MI 48109-2110

[2]Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina Del Rey, CA 90292

Abstract

The domains that computer-generated forces address (such as tactical flight) are more complex than have generally been used in artificial-intelligence research. A particular characteristic of this complexity is that a reasonable agent must attend to a large number of goals at the same time. Moreover, some of these goals are independent, while others interact with each other in a variety of ways. This research focuses on a number of issues involved in representing, reasoning about, and learning about such complex goal structures. We discuss a number of approaches that we have examined within the framework of the TacAir-Soar system.

The Soar-IFOR project aims to build believable agents for tactical air simulation. We have constructed a system, called TacAir-Soar, that embodies a large amount of knowledge for carrying out tactical naval air missions (Jones, Tambe, Laird, & Rosenbloom, 1993; Rosenbloom et al., 1994). In the course of our research, we have developed a large ontology of the knowledge required to generate human-like behavior in flight simulation. This includes knowledge about mission goals, doctrine, equipment specifications, survival, situational awareness and interpretation, cooperation, and other aspects of the task. Although each of these types of knowledge is relatively independent, their impact on behavior is highly interdependent.

This paper investigates various representations for sets of interacting goals that arise from such a complex knowledge base. We have identified five issues that we wish to address in our examination of the candidate approaches. First, it appears to be necessary to represent agent goals as a *forest* of interacting goal hierarchies. Second, existing goal-driven systems are not designed for such a goal representation, so we must find an appropriate mapping between agent goals and the types of goals that current architectures for intelligence allow (e.g., we want the architecture to do as much maintenance of goals as possible). Third, the agent must reason about how well different actions achieve combinations of goals. Fourth, the ideal knowledge representation should facilitate effective learning within the architecture. Finally, the representation should also allow the knowledge base to be updated by subject-matter experts and knowledge engineers with a minimum of effort.

An example from the tactical flight domain

To illustrate this complexity of knowledge, consider a situation where an F14 pilot has just launched a medium-range, radar-guided missile. At this point, the pilot has a number of active goals, such as surviving, accomplishing a specified mission, destroying the target, achieving another missile shot, maintaining situational awareness, and supporting the launched missile. A subset of these goals appears in Figure 1. Some of these goals have a direct hierarchical relationship (e.g., intercepting a target and achieving proximity to it), while others are relatively independent of each other (e.g., achieving proximity to a target and employing weapons). In response to this host of goals, there are a number of candidate actions the pilot could consider. However, these goals constrain and sometimes even conflict with each other, so it does not always suffice for the pilot to select an action that addresses only a subset of his or her goals.

In this case, the pilot may wish to decrease closing velocity to the target in order to increase chances of survival and to achieve another missile shot. One possible action would be to turn away from the target, but this would violate the goals of maintaining a radar lock and supporting the launched missile because the pilot's radar would no longer be illuminating the target. Another option would be to reduce speed by reducing thrust. This has the tradeoff of reducing the F14's energy, which could become important later in the engagement. Other possible actions would be to reduce speed by gaining altitude, or reduce closing velocity by turning part way away from the target (as in an "f-pole" maneuver). The amount of altitude change or f-pole turn would depend on other aspects of the current situation, such as the gimbal limits of the radar.

Issues for constructing an intelligent agent

Our approach to simulation is to apply state-of-the-art artificial-intelligence (AI) technology to create individual intelligent participants for simulated engagements. Unfortunately, existing AI systems that generate behavior are not well suited to the demands of knowledge-rich tasks with interacting goals. In general, AI systems only focus on one goal at a time, or at best allow a single hierarchy of simultaneous goals. However, some of the goals in the current domain are hierarchical in nature, while others clearly are not. Even the non-hierarchical goals interact and must be taken into account when generating behavior. In essence, it appears that the best representation of goal knowledge for this domain consists of a *set* of interacting goal hierarchies.

This is not to say there has been no research on planning to address unordered interacting goals. For example, Chapman (1987) presents a complete and correct planning method for arbitrary goal combinations, but it works in restricted domains, and it relies on search-intensive planning, rather than real-time behavior generation. Cohen, Greenberg, Hart, and Howe (1989) and Veloso (1989) have suggested methods for conjunctive goal planning in real time by storing preplanned episodes or using intelligent heuristic search. These are alternatives to the approach presented here, and we plan to examine the tradeoffs between various approaches in the future.

Rather than committing to a single potential solution, we are evaluating a number of different approaches both to the representation of goals within an agent, and its mechanisms for reasoning about interactions between goals. All of our efforts have been developed with variations of the TacAir-Soar agent (Jones et al., 1993; Rosenbloom et al., 1994), which is

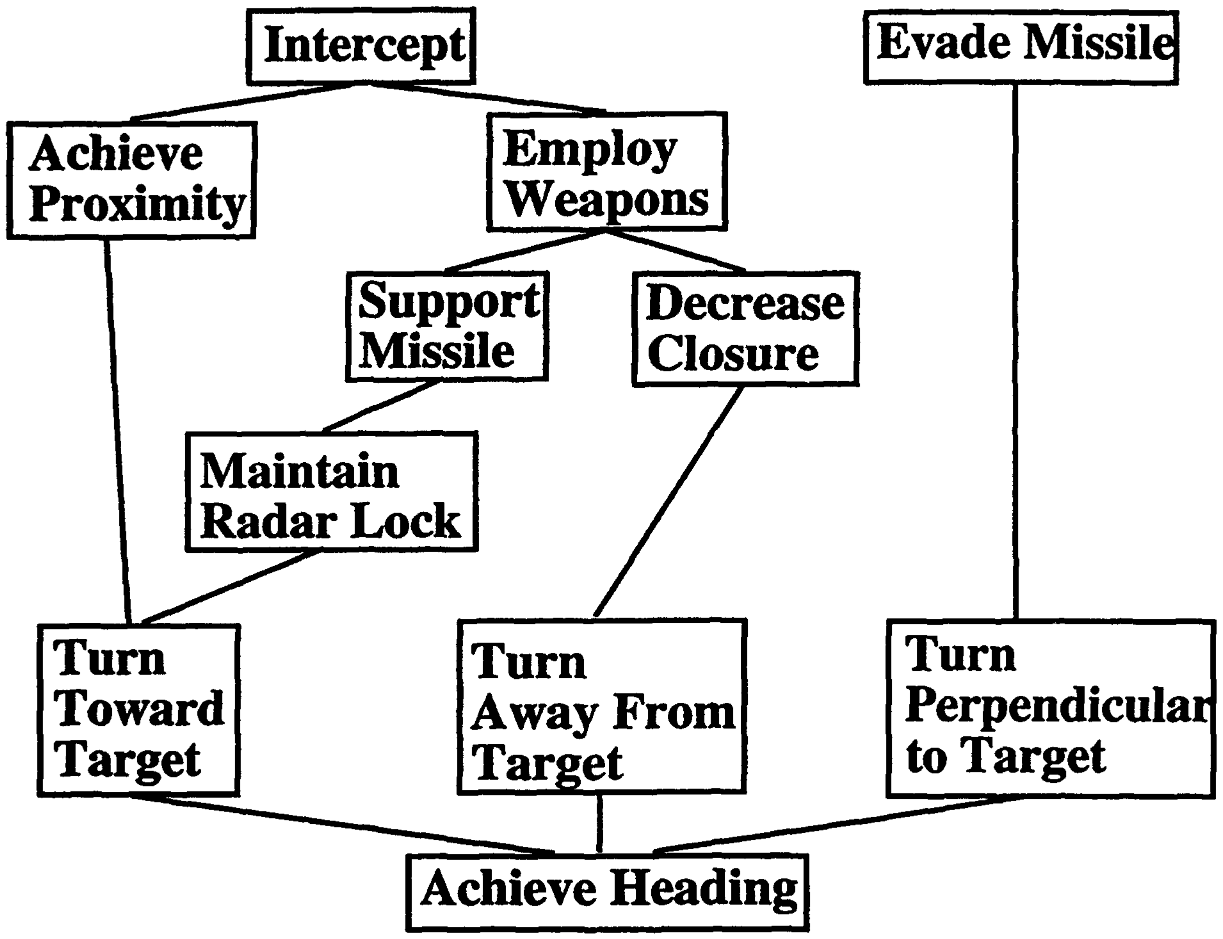


Figure 1. A subset of concurrent, interacting goals in the tactical air domain.

implemented within the Soar architecture for cognition (Rosenbloom, Laird, Newell, & McCarl, 1991).

Mapping agent goals to architectural goals

As the previous section illustrated, agent goals are best represented as a set of goal hierarchies. However, traditional AI systems do not encourage this type of representation. For example, architectures such as Soar, Prodigy (Minton et al, 1989), and Theo (Mitchell et al., 1991) make goals a first-class object type, and they include specific mechanisms for representing, posting, and learning about goals. But these goals can only be expressed easily in a single stack or hierarchy. To overcome this limitation, alternative goal representations can be used, such as encoding goals as part of the agent's current state description. However, this type of representation precludes using many of the mechanisms the architecture provides directly to support goals. Thus, we are left with the question of how the *agent goals* we wish to represent can or should be mapped to *architectural goals* that the overall system supports.

The initial design of TacAir-Soar takes a mixed approach to the mapping between agent and architectural goals. Some of the agent goals are represented explicitly as architectural goals, whereas others appear as implicit goals in the agent's situation representation. The explicit goals map directly onto Soar's goal stack, and they benefit from Soar's goal maintenance and learning mechanisms. In contrast, implicit goals are recorded along with other descriptions of the agent's current state, such as it's vehicle type, current speed, missile status, etc. Some implicit goals (e.g., survival) are not represented at all, and are simply assumed to exist, and the behavior-generation rules take them into account even though they are not represented explicitly. In the above example, survival, maintaining situational awareness, and decreasing closure are implicit goals, whereas destroying the target and supporting the missile are mapped to architectural goals. In many ways, this is an *ad hoc* solution, but it allows the system to generate reasonable behavior by using architectural mechanisms to support the explicit goals while still allowing the implicit goals to modify behavior when appropriate. Thus, this representation works well for generating behavior, but difficulties arise when the system must learn to adapt that behavior. For example, there is no easy way for the system to detect that maintaining situational awareness sometimes conflicts with evading a missile.

In response to this problem, we have investigated two alternative approaches to mapping goals. In one approach, *all* agent goals are mapped into the architectural goal hierarchy, collapsing the agent's forest of goals into a single stack. The alternative approach is to map *none* of the agent goals into the architectural goal hierarchy. In this case, all reasoning takes place in the service of a single architectural goal, and all other goals appear as descriptions of the agent's current situation. There are a number of tradeoffs between these two approaches involving the automatic mechanisms for maintaining a goal stack and learning, and the flexibility of the representation of goals and knowledge about goals in terms of expressive power and ease of maintenance. For example, when mapping all agent goals to architectural goals, the current forest of goals must be collapsed into a single hierarchy. This new hierarchy dynamically imposes a syntactic parent-child relationship on some goals even when such a relationship does not exist semantically. For example, evading a missile might be assigned as a child of employing weapons, even though the goals do not really depend on each other.

This resulting hierarchy represents a single total ordering on the normally partially ordered goals, which can lead to difficulties in maintaining the goal stack. In the above example, if the goal to employ weapons goes away, the goal to evade a missile will also be popped from the stack, because it was arbitrarily set up as a child of the goal to employ weapons. On the other hand, because all of the goals are mapped to architectural goals, this version of the system can take better advantage of built in mechanisms for detecting and implementing learning opportunities. The architectures for intelligence that we have mentioned generally learn about relationships *across* architectural goals, but not *within* architectural goals. If all the reasoning takes place within a single architectural goal, no learning can take place.

Our experiences with various representations for goals have also led us to consider alternatives for expanding architectures such as Soar, so that it can explicitly represent sets of goal hierarchies rather than just a single hierarchy. If this effort is successful, it should provide us with all of the advantages of both of the extreme approaches mentioned above, because all agent goals would map directly to architectural goals in a simple manner.

Reasoning about interactions

In addition to an appropriate representation for goals, the agent must contain mechanisms for reasoning about the way goals influence each other. There are two general cases that we consider here. Two goals *interact* when they can be achieved or maintained concurrently, but they each constrain the behaviors that are appropriate. For example, in Figure 1, the agent can reduce closing velocity to its target while maintaining a radar lock by turning just until the target is on the edge of the radar. Different behaviors would be appropriate if these goals were being addressed independently. In contrast, some goals are simply impossible to achieve or maintain at the same time. In this case, we say the goals *conflict* with each other. Again referring to Figure 1, the agent cannot always maintain a radar lock if it is busy evading a missile. Thus one or the other goal must be suspended temporarily or ignored completely.

Each of the system variations we have explored addresses goal interactions and conflicts. In one of our approaches, interactions between goals are represented implicitly within the proposal conditions for actions. For example, an agent might propose the action of maintaining radar lock on a target *unless* there is an incoming threat that needs to be evaded. An alternative approach involves explicitly representing the interaction between goals, so the agent can reason about when to suspend goals or attend to multiple goals. In this case, the agent proposes the goals of maintaining radar lock and evading a missile independently, and separate reasoning determines which set of actions addresses these goals in the best way. For example, the agent may decide to evade because survival is a high-priority goal.

The agent currently makes these decisions with built-in arbitration knowledge about which goals interact with each other. However, a final important issue concerns how the agent would *learn* such knowledge with experience. We see a number of advantages from the ability to identify and learn about goal interactions and conflicts. If the agent finds itself in an unexpected situation, it should have the flexibility to generate reasonable behavior by evaluating the effects of different actions in light of the current set of goals. In addition, the knowledge acquisition task can be made easier if an agent programmer does not have to anticipate all the interactions and conflicts that may arise when new goals are added. Finally, if the system can detect interactions that human experts have not encountered (e.g., when testing new types of technology), the system may be able to discover new tactics for satisfying particular sets of goals.

Currently, none of our agent implementations detect or learn about goal interactions on their own. However, in developing alternative frameworks and goal representations, we have identified some approaches that may be useful in supplementing TacAir-Soar with this ability. As an example, suppose the system knows about the goals to evade threats and to maintain radar lock, but it has no knowledge about how these goals conflict. The system's first task is to detect the conflict. This occurs when it proposes actions to come to two different headings. This will cause an *impasse* in the Soar architecture, which identifies an opportunity to learn. Next, the system can plan by predicting outcomes of various actions, thus deciding which goals are more important to achieve, and which can be suspended temporarily. For example, the system may discover by mental simulation that it will be destroyed if it does not evade an incoming threat. Thus, the goal to evade should take precedence. In other situations, it may be more important to maintain the radar lock (e.g., the agent may have launched its own missile). Goal interactions will be handled in a manner similar to goal conflicts, except the system will have to be supplemented with extra evaluation knowledge so that it can appropriately measure the partial satisfaction of multiple goals.

Summary

There are a number of important issues involved in handling interacting and conflicting goals to generate reasonable behavior in a complex domain. Perhaps foremost are the facts that an intelligent system must be able to represent and reason about multiple concurrent goal hierarchies, and traditional goal representations in existing AI systems are inadequate. Given an appropriate goal representation, an agent must also be able to reason effectively about the possible interactions and conflicts between goals, producing the best behavior given all the various constraints. Finally, intelligent agents must eventually be able to acquire knowledge about interactions and conflicts automatically, so that the agent can behave flexibly and knowledge from subject-matter experts can be encoded without getting lost in tiny details.

We have experimented with a variety of representations for concurrent goal hierarchies, and attempted to fit these representations nicely into an existing AI architecture. We have been successful in this effort, but we have also discovered possible opportunities for improving the architecture itself. Although we have not yet solved all the problems with detecting and learning about goal interactions, our efforts so far have helped us identify how and where learning might occur within the existing TacAir-Soar system. Our current efforts involve refining our evaluation of the best representation for goals and implementing our ideas for learning. In addition, other researchers have investigated the issue of real-time planning for interacting goals (Cohen et al., 1989; Veloso, 1989). Although this work makes slightly different assumptions about the demands of the domain and real-time behavior generation, we hope to evaluate some of their ideas in the context of the TacAir-Soar system.

Acknowledgements

This research has benefited from the authors' interactions with all of the members of the Soar-IFOR group at the University of Michigan, University of Southern California, and Carnegie Mellon University. The members of BMH, Inc. have proved invaluable as subject-matter experts. The research is supported by contract N00014-02-K-2015 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Research Laboratory.

References

Chapman, D. (1987). Planning for conjunctive goals. In *Artificial Intelligence, 32*, 333-377.

Cohen, P. R., Greenberg, M. L., Hart, D. M., & Howe, A. E. (1989). Understanding the design requirements for agents in complex environments. *AI magazine, 10*(3), 32–48.

Jones, R. M., Tambe, M., Laird, J. E., & Rosenbloom, P. S. (1993). Intelligent automated agents for flight training simulators. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation* (pp. 33–42). Orlando, FL.

Minton, S., Knoblock, C. A., Kuokka, D. R., Gil, Y., & Carbonell, J. G. (1989). *Prodigy 2.0: The manual and tutorial.* Technical report no. CMU-CS-89-146, School of Computer Science, Carnegie Mellon University.

Mitchell, T. M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., & Schlimmer, J. (1991). Theo: A framework for self-improving systems. In K. VanLehn (Ed.), *Architectures for intelligence: The 22nd Carnegie Mellon Symposium on Cognition.* Hillsdale, NJ: Lawrence Erlbaum.

Rosenbloom, P. S., Johnson, W. L., Jones, R. M., Koss, F., Laird, J. E., Lehman, J. F., Rubinoff, R., Schwamb, K. B., & Tambe, M. (1994). Intelligent automated agents for tactical air simulation: A progress report. In *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation.* Orlando, FL.

Rosenbloom, P. S., Laird, J. E., Newell, A., & McCarl, R. (1991). A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence, 47*, 289–325.

Veloso, M. M. (1989). *Nonlinear problem solving using intelligent casual commitment.* Technical Report no. CMU-CS-89-210, School of Computer Science, Carnegie Mellon University.

Biographies

Randolph M. Jones received his Ph.D. in Information and Computer Science from the University of California, Irvine, in 1989. He is currently an assistant research scientist in the Artificial Intelligence Laboratory at the University of Michigan. His research interests lie in the areas of intelligent agents, problem solving, machine learning, and psychological modeling.

John E. Laird is an associate professor of Electrical Engineering and Computer Science and the director of the Artificial Intelligence Laboratory at the University of Michigan. He received his B.S. degree in Computer and Communication Sciences from the University of Michigan in 1975 and his M.S. and Ph.D. degrees in Computer Science from Carnegie Mellon University in 1978 and 1983, respectively. His interests are centered on creating integrated intelligent agents (using the Soar architecture), leading to research in problem solving, complex behavior representation, machine learning, and cognitive modeling.

Milind Tambe is a computer scientist at the Information Sciences Institute, University of Southern California (USC) and a research assistant professor with the computer science department at USC. He completed his undergraduate education in computer science from the Birla Institute of Technology and Science, Pilani, India in 1986. He received his Ph.D. in 1991 from the School of Computer Science at Carnegie Mellon University, where he continued as a research associate until 1993. His interests are in the areas of integrated AI systems, and efficiency and scalability of AI programs, especially rule-based systems.

Paul S. Rosenbloom is an associate professor of computer science at the University of Southern California and the acting deputy director of the Intelligent Systems Division at the Information Sciences Institute. He received his B.S. degree in mathematical sciences from Stanford University in 1976 and his M.S. and Ph.D. degrees in computer science from Carnegie-Mellon University in 1978 and 1983, respectively. His research centers on integrated intelligent systems (in particular, Soar), but also covers other areas such as machine learning, production systems, planning, and cognitive modeling. He is a Councillor of the AAAI and a past Chair of ACM SIGART.

Knowledge Acquisition and Knowledge Use in a Distributed IFOR Project

Frank Vincent Koss
Artificial Intelligence Laboratory
University of Michigan
1101 Beal Avenue
Ann Arbor, MI 28109-2110
koss@umich.edu

Jill Fain Lehman
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
jef@cs.cmu.edu

Abstract

A fundamental goal of the IFOR/WISSARD project is the creation of autonomous, intelligent agents that can participate in computer simulations of battle for training and gaming purposes. The creation of such an agent has many of the same requirements as constructing an expert system. In particular, the designers face the enormous task of acquiring, encoding, and refining the knowledge that defines the agent's desired behavior. The knowledge must be drawn from many sources, e.g. subject matter experts (SMEs), training manuals and other texts, observation, experimentation, etc. This is usually done by many people whose raw materials must then be represented as a coherent specification that designers can use for constructing the agent, communicating among themselves, and, ultimately, describing the causes and rationales for the agent's behavior to others. In any case, this is not a simple task, but in the case of the TacAir-Soar project, the difficulty is increased by the geographical distribution of the project's members.[1] Our solution to this problem is an electronic, multi-layer hypertext document called the TacAir-Soar Description Document (TDD) which is implemented within the National Center for Supercomputing Applications' (NCSA's) Mosaic. This document allows its viewers to obtain information about the domain in plain English, about the agent in terms of its structures and behaviors, and about the actual code that implements the agent.

Introduction

The TacAir-Soar project requires information from many sources. The sources used to date include interviews with SMEs, electronic mail messages, training manuals, telephone conversations, and observation of fighter pilots during simulated engagements. To be useful, all of this information must be combined into a single, organized repository that can be accessed by all members of the project at each of the three sites involved (University of Michigan, University of Southern California/Information Sciences Institute, and Carnegie Mellon University). Additionally, there needs to be some way to show the influences of various pieces of knowledge on the design and development of the agents. Without this, there is no way for SMEs to validate the relationship between the domain knowledge they provide and the agent's behavior.

Given these requirements, the NCSA's Mosaic system was chosen as the application within which to create a document that would combine domain knowledge with agent implementation in a coherent manner that could be accessed across the Internet. The TacAir-Soar Description Document has three layers, corresponding to the three levels of specification we use to discuss agent behavior. The top layer of the document reflects the *knowledge level*, an English description of the air-to-air combat domain. This is a level of specification that is concerned with the knowledge of objects, actions, and relations in the domain independent of any particular computational implementation of that knowledge. Although the top layer of the TDD presents a coherent view of the domain by integrating across particular instances of knowledge acquisition, information gathered from any of the sources listed above (interviews, electronic mail, etc.) is also linked[2] to the top layer in its raw form to allow for traceability.

Items in the top layer of the TDD may also have links into the next layer of the document which describes the agent's structures and behaviors at the level of the *problem space computational model* (PSCM). The PSCM-level is a description of the agent's behavior in terms of an

[1]For a description of the TacAir-Soar project, see this volume, [Rosenbloom94].

[2]In a hypertext document, *links* are portions of the text that are highlighted in some way and are associated with another document. If a link is selected, the document to which it refers is displayed.

abstract model of the Soar architecture, independent of the particular implementation of the architecture in C. Each PSCM-level document links to the *symbol level* representation of the agent, i.e. to a matching Soar code file in the third layer of the TDD. The code files in the third layer are the actual files that are loaded when an agent is created, so they are always current. Because the layers are linked, a user can work up or down through the hierarchy. Working downward means beginning with the description of a particular concept and following it through the layers to its implementation. Working upward means moving from the agent code through the layers to find the justification for a particular structure or behavior.

Platform and Justification

Mosaic is a hypermedia browser distributed by the NCSA. It allows a user to view documents that contain plain text, formatted text, PostScript, images and diagrams, audio, and digitized video. When combined with servers that use the HyperText Transport Protocol (HTTP) [Berners-Lee92], Mosaic can be used to view documents that are located throughout the world on machines that are connected to the Internet.

These features mean that Mosaic has many advantages. First, because it is already written and has a large number of users world-wide, we do not need to spend time developing or maintaining our own tool. The large base of users also means that tools that support the authoring of documents, such as editors and translators, are readily available. Second, Mosaic is in the public domain so there is no monetary cost associated with the TDD's development or use. Third, Mosaic runs on many Unix workstations and on the Macintosh, so all group participants are able to access the documentation regardless of the machine they normally use. Fourth, through the use of a server, all members can access the same copies of the document at all times.[3] Changes are immediately accessible to all, reducing the chance of out-of-date documentation causing confusion. Finally, because of the multimedia capabilities, we can include such items as diagrams for describing tactics and maneuvers, images of equipment, and unmodified electronic mail messages. This flexibility allows us to use the most appropriate means to store and convey information.

The Knowledge Level

The top layer of the TDD describes the domain of air-to-air combat at the knowledge-level, i.e. independent of any particular computational implementation of the domain. The information contained in this layer is of a general nature and is obtained from a number of sources. All source material is kept and is referenced by the domain description to allow for traceability. Interviews with pilots are videotaped whenever possible. (These are turned into documents that are considered the source of this information. The videotapes themselves are also kept.) Telephone interviews are also turned into source documents. Electronic mail messages are given a standard identifying header. For information from manuals and books, a bibliography-style entry is kept.

Knowledge from these disparate sources is organized into a coherent whole by the use of a topical tree-like structure and an index by topic area. Reorganizing the information in this way has a number of benefits. Combining from multiple sources on the same topic quickly reveals contradictions and missing or unclear information. Further, as detail is added to specific topics, the new information is near the older, more general knowledge and so is easy to locate. Since all of the organized documents are given similar formats, they are easier to browse than the various source documents. Finally, browsing the information topically is generally the easiest method for users.

The PSCM Level

The layer below the knowledge level gives a description of the agent's behavior in terms of an abstract model of the Soar architecture, independent of the particular implementation of the architecture in C. The PSCM is the basis of Soar and is the common view shared by the project participants. It is a view of problem solving behavior in which the agent pursues its goals by applying operators to the current state thereby deriving a new state in an iterative process until the goal state is achieved. Thus, this level of description maps domain knowledge into the form of goals, operators, and state information.[4] By following links between the knowledge-level documents and those at the PSCM level, the effect of the knowledge on the structure of the agent can be determined. In addition, areas of the agent that would be affected by additional information in a particular domain area can be found. Since the PSCM is a specific instance of many ideas that are common within AI, this level of representation may be accessible to many subject matter experts as well.

[3] Anyone with access to Mosaic can view our documentation with the URL http://krusty.eecs.umich.edu/ifor.

[4] An understanding of Soar and the PSCM can be gained from [Newell90] and [Rosenbloom93].

The Symbol Level

There is a one-to-one mapping between the documents at the PSCM level and the Soar agent code, the lowest layer of the TDD. These files can be viewed to see how the definition of a problem space or operator was realized in code that executes within the implementation of the architecture. Since the PSCM is an abstraction, design choices at that level may have many realizations in the code. By separating the code level into its own layer, we also separate the general constraints on that realization (e.g. that it should be an operator rather than state) from the realization itself. By linking the two layers we maintain a record of the origin of our coding choices.

Example of Use

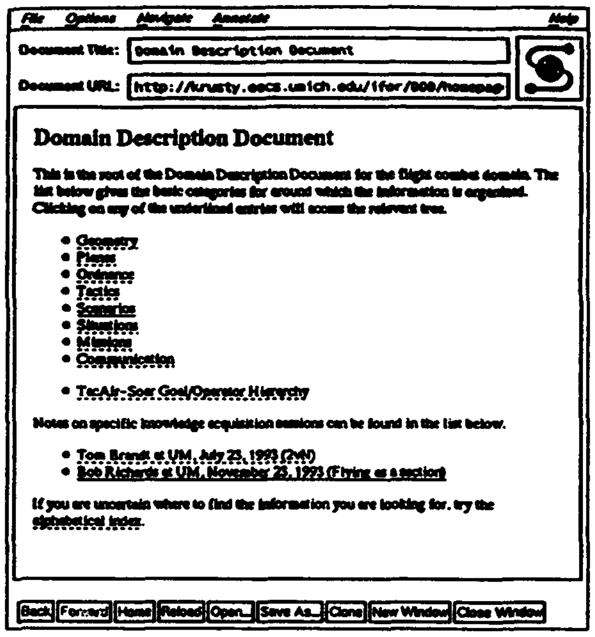
Figure 1 shows the root of documents which make up the knowledge level. Underlined words and phrases indicate links to other documents. The links Geometry through Communication connect to parts of the knowledge-level document hierarchy that cover those topics. TacAir-Soar Goal/Operator Hierarchy links to the root document of the PSCM level. Tom Brandt at UM, July 23, 1993 (2vN) links to a source document that was created from a videotaped interview.

Following a piece of the topical organization downward, figure 2 shows the document that is linked to by the Geometry link of figure 1. This document, still part of the knowledge level, has a figure that shows the terms used to describe the geometry of two aircraft. The link Target Aspect then leads to the document in figure 3, which gives a narrative description of target aspect. This description ends with a link that goes to a knowledge level document that covers the related topic of lateral separation. Below the description of target aspect are links to other relevant knowledge-level concepts, as well as a link to the source for this document's information.

At the bottom of figure 3 there are links to the three documents in the PSCM level of the TDD that involve target aspect. Following one of these, Cut-to-ta, leads to the document in figure 4, which describes the operator that is used to cause the agent to turn its aircraft in order to achieve a desired target aspect. Finally, following the link /top-ps/.../cut-to-ta.soar into the symbol level of the TDD displays the Soar productions that implement the PSCM-level operator, as shown in figure 5.

Conclusions

Initial knowledge acquisition results in changes to the domain level of the document. This information is in a form that both project members



Document Title: Domain Description Document

Document URL: http://krusty.eecs.umich.edu/ifor/DDD/homepag

Domain Description Document

This is the root of the Domain Description Document for the flight combat domain. The list below gives the basic categories for around which the information is organized. Clicking on any of the underlined entries will access the relevant tree.

- Geometry
- Planes
- Ordnance
- Tactics
- Scenarios
- Situations
- Missions
- Communication

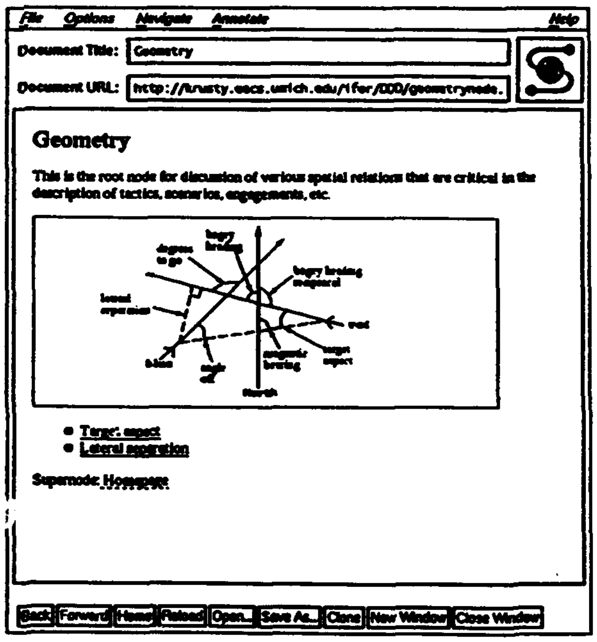
- TacAir-Soar Goal/Operator Hierarchy

Notes on specific knowledge acquisition sessions can be found in the list below.

- Tom Brandt at UM, July 23, 1993 (2vN)
- Bob Richards at UM, November 23, 1993 (Flying as a section)

If you are uncertain where to find the information you are looking for, try the alphabetical index.

Figure 1: Root of the Knowledge Documentation.



Document Title: Geometry

Document URL: http://krusty.eecs.umich.edu/ifor/DDD/geometrynode.

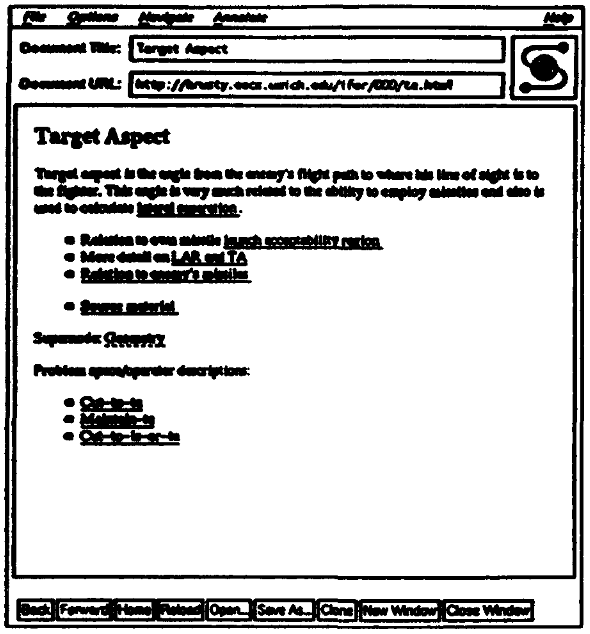
Geometry

This is the root node for discussion of various spatial relations that are critical in the description of tactics, scenarios, engagements, etc.

- Target aspect
- Lateral separation

Supernode: Homepage

Figure 2: The Geometry Knowledge Document.


File Options Navigate Annotate Help
Back Forward Home Reload Open... Save As... Clone New Window Close Window

Document Title: Target Aspect

Document URL: http://krusty.eecs.umich.edu/ifor/IDD/ta.html

Target Aspect

Target aspect is the angle from the enemy's flight path to where his line of sight is to the fighter. This angle is very much related to the ability to employ missiles and also is used to calculate lateral separation.

- Relation to own missile launch acceptability region
- More detail on LAR and TA
- Relation to enemy's missiles

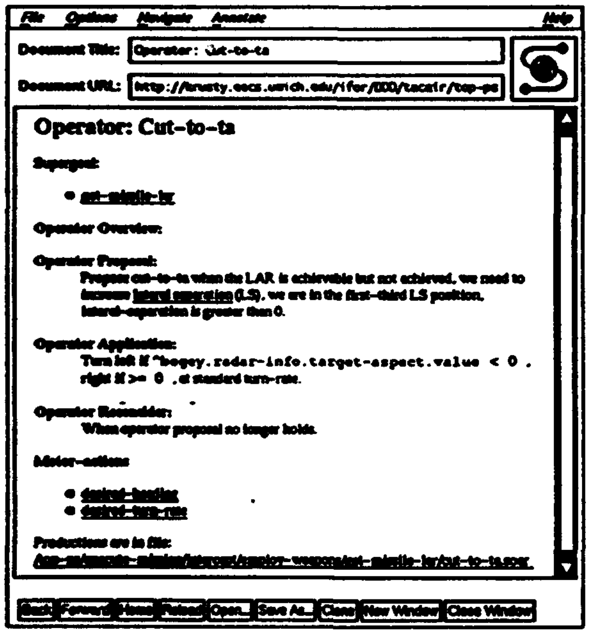
- Source material

Supernode: Geometry

Problem space/operator descriptions:

- Cut-to-ta
- Maintain-ta
- Cut-to-ls-or-ta

Figure 3: The Target Aspect Knowledge Document.


File Options Navigate Annotate Help
Back Forward Home Reload Open... Save As... Clone New Window Close Window

Document Title: Operator: Cut-to-ta

Document URL: http://krusty.eecs.umich.edu/ifor/IDD/tacair/top-ps

Operator: Cut-to-ta

Supergoal:

- get-missile-lar

Operator Overview:

Operator Proposal:
Propose cut-to-ta when the LAR is achievable but not achieved, we need to increase lateral separation (LS), we are in the first-third LS position, lateral-separation is greater than 0.

Operator Application:
Turn left if ^bogey.radar-info.target-aspect.value < 0 , right if >= 0 , at standard turn-rate.

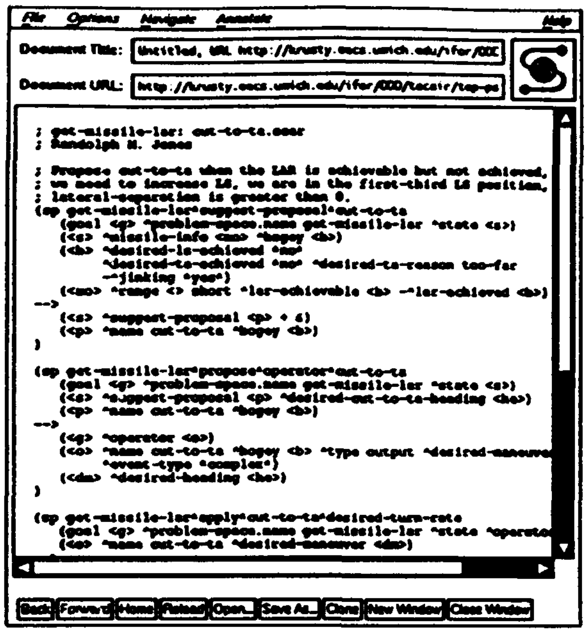
Operator Reconsider:
When operator proposal no longer holds.

Motor-actions

- desired-heading
- desired-turn-rate

Productions are in file:
[illegible]/employ-weapons/get-missile-lar/cut-to-ta.soar

Figure 4: The Cut-to-ta PSCM document.


File Options Navigate Annotate Help
Back Forward Home Reload Open... Save As... Clone New Window Close Window

Document Title: Untitled, URL http://krusty.eecs.umich.edu/ifor/IDD

Document URL: http://krusty.eecs.umich.edu/ifor/IDD/tacair/top-ps

```
; get-missile-lar: cut-to-ta.soar
; Randolph M. Jones

; Propose cut-to-ta when the LAR is achievable but not achieved,
; we need to increase LS, we are in the first-third LS position,
; lateral-separation is greater than 0.
(sp get-missile-lar*suggest-proposal*cut-to-ta
   (goal <g> ^problem-space.name get-missile-lar ^state <s>)
   (<s> ^missile-info <mo> ^bogey <b>)
   (<b> ^desired-ls-achieved *no*
        ^desired-ta-achieved *no* ^desired-ta-reason too-far
        -^jinking *yes*)
   (<mo> ^range <> short ^lar-achievable <b> -^lar-achieved <b>)
-->
   (<s> ^suggest-proposal <p> + &)
   (<p> ^name cut-to-ta ^bogey <b>)
)

(sp get-missile-lar*propose*operator*cut-to-ta
   (goal <g> ^problem-space.name get-missile-lar ^state <s>)
   (<s> ^suggest-proposal <p> ^desired-cut-to-ta-heading <he>)
   (<p> ^name cut-to-ta ^bogey <b>)
-->
   (<g> ^operator <o>)
   (<o> ^name cut-to-ta ^bogey <b> ^type output ^desired-maneuve
        ^event-type *complex*)
   (<dm> ^desired-heading <he>)
)

(sp get-missile-lar*apply*cut-to-ta*desired-turn-rate
   (goal <g> ^problem-space.name get-missile-lar ^state ^operato
   (<o> ^name cut-to-ta ^desired-maneuver <dm>)
```

Figure 5: The Soar productions for cut-to-ta.

and non-project members can access and evaluate. Designers can then decide how to realize the new information within the agent. This discussion tends to take place at the abstract level of the PSCM, and is subsequently recorded as additions or changes to the PSCM layer. Once the PSCM design is complete, coding can begin. If there are decisions that cannot be made unambiguously at the symbol level, pointers back through the PSCM can help direct attention to other parts of the code that may be relevant.

The system we have described has proven to work well. Both project members and domain experts are able to access the information. The ability to include images and diagrams has proven to be very useful.

There are a number of limitations which need to be explored in more detail. The question of centralized vs. de-centralized control of the document has become increasingly important, as is usual for any dynamically changing resource. Centralized control has thus far ensured that all have access to consistent information. However, this has proven to be a bottleneck in the process of adding new information. Also at issue are the possible roles of digital libraries in expanding the type of information that can be included in the "document" (e.g. video of experiments rather than just transcriptions) and of the Internet in making such a document more widely available.

Acknowledgements

This research was supported under subcontract to Carnegie Mellon University from the University of Michigan and at the University of Michigan as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). Critical support has been provided by Tom Brandt, Bob Richards, and Ed Harvey of BMH Inc.

References

[Berners-Lee92] Berners-Lee, T.J, Cailliau, R., and Groff, J.F., The World-Wide Web, Computer Networks and ISDN Systems 25 (1992) 454-459. Noth-Holland.

[Newell90] Newell, A., Unified Theories of Cognition, Harvard Press, Cambridge, MA, 1990.

[Rosenbloom93] Rosenbloom, P. S., Laird, J. E., and Newell, A., The Soar Papers: Research on Integrated Intelligence, MIT Press, Cambridge, MA, 1993.

[Rosenbloom94] Rosenbloom, P. S., Johnson, W. L., Jones, R. M., Koss, F., Laird, J. E. Lehman, J. F., Rubinoff, R., Schwamb, K. B., and Tambe, M., Intelligent automated agents for tactical air simulation: a progress report. In Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation, 1994.

Biographies

Frank V. Koss is a systems research programmer in the Artificial Intelligence Laboratory at the University of Michigan, where he is developing the interface between the Soar architecture and the ModSAF simulator. He received his BS in computer engineering from Carnegie Mellon University in 1991 and his MSE in computer science and engineering from the University of Michigan in 1993. He is a member of IEEE and AAAI.

Jill Fain Lehman is a research computer scientist in Carnegie Mellon's School of Computer Science. She received her B.S. from Yale in 1981, and her M.S. and Ph.D. from Carnegie Mellon in 1987 and 1989, respectively. Her research interests span the area of natural language processing: comprehension and generation, models of linguistic performance, and machine learning techniques for language acquisition. Her main project is NL-Soar, the natural language effort within the Soar project.

Coordinated Behavior of Computer Generated Forces in TacAir-Soar

John E. Laird, Randolph M. Jones, and Paul E. Nielsen
Artificial Intelligence Laboratory
University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110
laird@umich.edu

Abstract

The fielding of large numbers of autonomous computer-generated forces requires that these forces be able to coordinate their behaviors. Within the military, there are many levels of coordination, from the high-level management of a theater of war, down to the low-level interactions of individual soldiers. TacAir-Soar represents a data point at this low level, where individual fighter planes must fly together in sections with support from a air intercept controller. In this paper we analyze the types of coordinated behavior required to make TacAir-Soar a realistic model of human behavior, the methods that our agents employ to coordinate their behavior, and finally, the constraints coordination places on the design of computer-generated forces.

Introduction

One of the ultimate goals of research in computer-generated forces is to populate simulated battlefields with automated intelligent agents[1] which behave as humans would on a real battlefield. Although we can make progress by creating more and more individual agents, we will still be far short of modeling human behavior unless we create agents that coordinate their behavior.

The reasons for coordinating the behavior of individuals are obvious. A single unit has only limited ability to sense its environment directly, and limited ways in which it can act on its environment. Through coordination of sensing, multiple agents can share their knowledge about the environment, thus making whatever action they take more effective. Through coordination of their actions, multiple agents can perform actions that no single agent can perform, such as creating diversions and supporting actions, or bringing to bear fire power that no single agent has alone. The problem is how to get many different agents, in different physical locations, with different models of the environment, with different physical abilities, and possibly different short-term goals, to work together to achieve the most effective results.

[1]Throughout this paper we will use the term *agent* to refer to a single computer-generated entity, such as a pilot of a fighter plane.

In the past, computer-generated forces have taken one of three approaches:

1. No coordination
   Many computer generated forces do not attempt to coordinate their behavior with any other forces. They have a specific mission that they are to execute, and they execute the mission independent of other friendly forces. In many semi-automated forces (SAFORs), it is left for an overseeing human to organize their behavior. Sometimes this requires that the human "micro-manage" the individual units, and in the heat of battle, the human can become overloaded.
2. Centralized control
   When tight coordination of behavior of a small unit is required, the common approach is to treat the aggregation as a single unit in terms of behavior. For example, individual tanks may be represented on the battlefield, but their behavior is organized into platoons and companies. Instead of attempting to represent the communication and coordination of the individual tanks, behavior is generated for the platoon (or company) as a whole and then specialized for the individual unit (a tank). Each unit does not independently reason about its behavior and there is no explicit communication between units.
3. Explicit command and control
   In a limited number of cases, computer-generated forces have generated explicit orders to lower-echelon forces, as in Eagle II [Powell and Hutchinson, 1993]. However, this did not include real-time interaction between independent units.

The conclusion is that only limited progress has been made in creating agents that coordinate their behavior in flexible ways. Unfortunately, solving the general problem of dynamically organizing multiple agents to maximize their coordination is an intractable problem. However, to create coordinated automated forces does not require a complete solution to this problem. We can limit ourselves to modeling the methods and practices currently used by military organizations. Within the military, the command structure is a relatively static hierarchy, where preplanning and training are used extensively to avoid the complexities, delays, communication difficulties, and possible confusion that can arise with dynamic reorganization or retasking of the participants. Also, much of the behavior is determined by predefined tactics and doctrine, which reduces the need for communication. This is not to say that such reorganizations and retaskings are not possible, it is just that they are held to a minimum, and are based on well defined procedures.

Thus, our goal is not to develop new forms of coordination and communication for the military (they have been working on this for thousands of years), but instead to create computer-generated forces that can participate in coordinated behavior within the limits (and breadth) of a military organization. Our goal is to identify the types of coordination and communication that must be supported by an intelligent agent and then examine how this impacts on the design of computer generated forces.

Our approach starts with individual units that independently reason about their own behavior and coordinate their behaviors using explicit communication as well as shared tactics and doctrine. We plan on using explicit command and control, but with the intent that it is ubiquitous and used more flexibly and robustly than has been demonstrated to date. Some of the advantages of this approach are as follows:

1. Coordinated behavior will be more realistic. Coordination based on communication will be explicit, require time to transmit and interpret, be open to mis-interpretation, jamming, etc. Coordination based on shared doctrine and tactics will obey doctrine, but it will also fail when the doctrine fails. In addition, by independently modeling each entity (instead of a group as a whole), it should make it easier to model doctrine where the individual unit or subgroup is expected to have initiative.

2. Coordinated behavior should scale up to higher levels of command. Instead of trying to create larger and larger aggregate forces that are centrally controlled, commanding agents are created (such as platoon, company, battalion commanders) whose purpose is to generate commands for lower levels and report back to higher levels.

3. Coordinated behavior should be easier for humans to understand because there will be explicit communication that can be observed.

4. Coordinated behavior between human and computer generated forces will be possible.

In this paper, we report on the first steps at coordinated behavior within automated forces by examining our implementation of the coordination required for two planes flying tactical air missions as a section. Of necessity, we have been studying low-level real-time coordination that arises during the execution of a specific mission. We have not studied the longer term coordination that is required at higher levels of the command hierarchy such as managing an air or ground campaign.

Our approach is to treat our work as a case study. We start by analyzing the coordination required for flying two planes in a section in our current implementation. Next we study the various methods that computer generated forces can use to obtain the knowledge required to coordinate their behavior. This leads to the main point of the paper which is to identify how coordination impacts the design of computer generated forces.

Example Scenario

The environment in which we are studying coordination is tactical air combat, as part of the Soar/IFOR component of WISSARD. The agents we are modeling include fighter planes, such as F-14's and MiG-29's, and air intercept controllers (AIC) in AWACS-like planes such as the E-2C. Our IFOR agents are built in TacAir-Soar [Rosenbloom *et al.*, 1994] within the Soar architecture [Laird *et al.*, 1987] and interact with the DIS world through ModSAF [Calder *et al.*, 1993]. Each agent is independently situated in its own vehicle (such as an F-14, an MiG-29, or an E-2C), and is restricted to perceiving what is available on its own vehicle's sensors. Our agents communicate via radio messages that approximate the messages sent by human pilots. In our current implementation, our agents perform 2v2 intercepts (as either red or blue, or both).

Consider the scenario in Figure 1 in which two blue fighter planes (F-14's) are flying together as a section in a combat-air patrol (CAP) protecting an aircraft carrier, with help from an air intercept controller on an E-2. The distances and sizes of planes are not to scale. Two red enemy enemy planes (MiG-29's) are coming in from the east to attack the aircraft carrier; posing a threat that the blue fighters must respond to. In the remainder of this section, we present the types of coordination implemented within TacAir-Soar

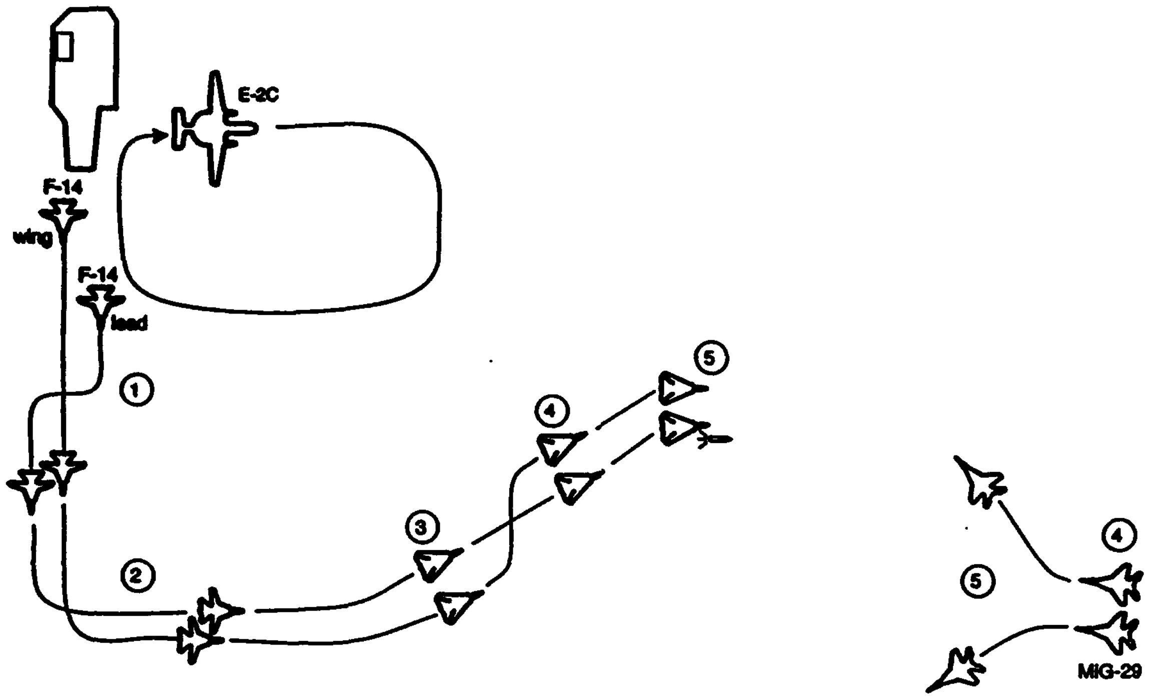


Figure 1: Example Scenario

using examples from this scenario as the blue and red fighters engage.

Flying as a Section

Historically, a section of two planes has been found to be the minimal effective fighting unit. A section consists of a lead and a wingman flying together on a joint mission. The tactical lead of the section directs the maneuvering of the section, either through his actions or through explicit communication to the wingman. The goal of the wingman is to stay in formation and support the activities of the lead (such as through manipulation of its radar). In some circumstances, the wingman will take over as lead (such as if the lead's equipment malfunctions, or the lead is out of missiles).

To be an effective section, the lead and wingman must coordinate their maneuvering, their sensing of the environment, their employment of weapons, and the organization of their section. Below is a detailed list of the behaviors that have been implemented in TacAir-Soar to coordinate behavior for beyond-visual-range engagements such as in Figure 1. These descriptions (and our implementations) are idealizations of the real behaviors, but capture much of the essence of the real behaviors.

Maneuvering

- **Joining up in Formation**
  If the planes of a section are split, possibly after taking off, or following an engagement, the planes must join up into a formation. It is the responsibility of the wingman to obtain the correct position and this is done using visual and radar cues without communication. However, in cases when the two planes are far apart, the wingman may request position information from the lead. Although flying into formation is primarily the responsibility of the wingman, if the lead is far ahead, the lead may maneuver, possibly employing a shackle turn, to allow the wingman to catch up as in position 1 of Figure 1.

- **Flying in Formation**
  A section of planes can fly in many different formations, such as defensive combat spread, offensive combat spread, fighting wing, cruise, or trail. When flying in formation, it is the responsibility of the wingman to maintain the appropriate position. In Figure 1, the F-14's are initially in a parade formation.

- **Changing Formation**
  The specific formation used by a section can change as the tactical situation changes. For example, a section might start off in a tight parade formation until it gets to its CAP station and then assume a defensive combat spread at in position 3 of the scenario. It maintains that formation until the later parts of an engagement when the planes are closing on an enemy, at which time they then move to an offensive

combat spread.

- **Coordinated Maneuvering**
  As the lead maneuvers, the wingman attempts to stay in formation as in position 3 of the scenario. However, for large turns, the section must perform special maneuvers or else the wingman will get out of formation. For example, when the lead wishes to turn 90 degrees toward the wingman, as in position 2, the lead will turn first and then the wingman will turn once the lead crosses behind him. Conversely if the lead wishes to turn away from the wingman, the wingman turns first. Other maneuvers include in-place turns and crossover turns.

- **Tactical Maneuvering**
  When engaging enemy planes, a section can use special maneuvers in order to improve the geometry of an attack, or to confuse an enemy. Example maneuvers include a pincer (and half pincer), when the two planes separate and then close on an enemy, and a post-hole, where the section flies in a trail maneuver and the lead plane flies in a circle (to defeat an expected missile and possibly confuse the enemy), giving up the lead to the second plane, which then presses the attack. In Figure 1, the red planes attempt to employ a pincer at position 5. In addition, a section of planes may perform defensive maneuvers together, such as jointly turning into the beam to break radar lock and avoid a missile. Finally, when attacking an enemy, the wingman will usually attempt to slide to the outside of the formation to give the lead better position for the attack, as in position 4 of the scenario.

Sensing

- **Radar**
  By coordinating their radars, two planes can cover more area. The details of the "radar contract" can be determined during the briefing before the actual mission. When planes do get contacts, they communicate the relevant position information. Planes can also request information if they have lost a contact.

- **Vision**
  Because it is sometimes difficult for a plane to detect enemy planes that are behind it, an important responsibility is to check the rear of the other plane. Another important use of vision is to identify unknown planes. Thus, a section may split up so that one plane can get a visual identification while the other is positioned for a shot if the plane is hostile. This type of within-visual-range coordination is not yet implemented in TacAir-Soar.

Employing Weapons

- **Targeting**
  If there are multiple groups of enemy planes approaching, the lead (possibly with the AIC) must determine which group to attack first and communicate this to the wingman.

- **Sorting**
  When a section engages multiple enemy planes, it is critical that the wingman and lead not waste missiles by shooting at the same plane. Thus, they must *sort* the enemy planes, possibly by range, altitude or azimuth, so they are targeting different planes. In general, the lead will take the plane that is the highest threat (usually the lead of the opposing section).

Controlling the Section

- **Changing the Lead**
  The role of the planes within a section can change if the wingman is in a better tactical situation, such as having more appropriate weapons or better situational awareness. When the wingman takes over, he must assume all of the responsibilities of the lead, and vice versa for the original lead.

Communicating Intent

- **Committing an Enemy**
  When an enemy plane has been identified as a bandit, and the commit criteria are reached, the lead will communicate the intent to intercept to the wingman.

Flying a Section with an AIC or GCI

Normally, a section of planes will have support from either an airborne or ground-based radar (GCI). These radars provide a much broader picture (approximately 250 nms for an E-2) and can detect attacking planes well before the section itself will see the attackers on its radars. Thus, for blue, the AIC provides position and identity information about other planes. The AIC can also provide engagement information and redirect the mission of a section, although this is not currently implemented in TacAir-Soar. For red, the GCI is more in control and may direct the tactics used by the section (this is also not implemented in TacAir-Soar).

Figure 2 is a dialogue produced between TacAir-Soar agents acting as an AIC (Kiwi) and a section of two F-14's (Hawk121 and Hawk122) as they engage a MiG-29 in a slightly different scenario than in Figure 1. Lines starting with ";" are comments and were not part of the communication. We have not attempted to duplicate the communication produced by humans exactly, but instead we have attempted to include the inter-

```
Kiwi: kiwi, hawk121 your bogey is at bearing 23 for 143 angels 8
    ; Each plane prefaces its communication with its call sign.
    ; Here Kiwi is giving the bearing (23 degrees), range (143 nms) and altitude (8,000 ft).
Hawk121: Roger
Kiwi: kiwi, Contact is a bandit
    ; Kiwi is confirming that the bogey is an enemy plane.
Hawk121: hawk121, Contact is a bandit
Hawk122: Roger
Hawk121: hawk121, Commit bearing 23 for 140 angels 8
    ; Hawk121 decides its commit criteria have are achieved and starts to intercept the bandit.
    ; Hawk121 uses the information from Kiwi to plot an intercept course
Kiwi: kiwi, hawk121 your bogey is at bearing 21 for 137 angels 8
    ; Kiwi periodically reports position information to the fighters.
Hawk121: Roger
Hawk122: Roger
Kiwi: kiwi, Bandit is closing on a hot vector
Hawk121: hawk121, Bandit is closing on a hot vector
Hawk121: hawk121, Go to defensive combat-spread formation.
    ; The section changes formation for the attack.
Kiwi: kiwi, hawk121 your bogey is at bearing 12 for 116 angels 8
Hawk121: Roger
Hawk122: Roger
Hawk121: hawk121, Bandit is closing on a hot vector
Hawk121: hawk121, Fox three
    ; Hawk121 fires a long-range missile and then performs an f-pole maneuver.
Hawk121: hawk121, Cranking right
```

Figure 2: Trace of communications between a F-14 section (Hawk121 and Hawk122) and an E-2 (Kiwi).

actions that are necessary for the planes to coordinate their behavior.

Methods for Coordination

For a section to coordinate its behavior, the individual agents must know many things. They must know the appropriate techniques and methods for maneuvering, sensing, employing weapons and controlling the section. They must also know the specific constraints under which the current mission is being flown, such as rules of engagement, commit criteria, and so on. During the mission, they must also build up their situational awareness, from their own sensors and through communication with others. Finally, they must coordinate their actions in the face of the world around them. These different types of knowledge are acquired at different times using the types of methods listed below.

Common Doctrine and Tactics

Doctrine and tactics specify methods and procedures for behaving in the world. This is similar to *social contracts*, where independent agents can create coordinated behavior by agreeing to behave in certain ways under specific circumstances [Shoham and Tennenholtz, 1992]. For example, drivers in the United States coordinate their behavior (and thus avoid accidents) by always driving on the right side of a street. Similarly, the lead and wingman have a division of labor so that they are not both trying to the same activity (such as maintain formation) at the same time.

From the perspective of coordination, common doctrine eliminates the need for communication (two cars passing each other do not need to negotiate which side they will pass), it allows an agent to predict the behavior of other agents without even knowing the exact identity of the agent, and it reduces the cognitive load on an agent because an agent does not have to plan out its behavior from first principles.

In TacAir-Soar, common doctrine and tactics are represented in Soar's long-term memory as rules (as is all long-term knowledge). This constitutes the vast majority of knowledge encoded in TacAir-Soar.

Mission Briefing

Before a mission, the participants are briefed on the tactical situation, their responsibilities, and often, the responsibilities of others.

The briefing helps establish specific operational parameters required for coordination, such as the specific partners of a section, their formations, the methods for communication (radio frequencies, call signs), the default radar contract, the default method for sorting bandits, any specific tactics the section plans to employ, and so on.

In TacAir-Soar, the mission and all information relevant to the current run is entered via an editor that is an extension of ModSAF. This includes the sides of the agents, the call sign of the agent, the type of airplane being flown, rules of engagement, the location of mission-relevant landmarks, and so forth. The information is the loaded into Soar's short-term memory, which makes it accessible to all of the rules in TacAir-Soar.

Observed Behavior

During a mission, the members of a section can directly observe each other's behavior. Thus, behavior alone can be a signal for coordinating behavior, as when a lead makes a small turn, without any explicit communication.

In TacAir-Soar, there is only limited use of coordination through observed behavior, with the wing responding to small turns of the lead being the best example.

Explicit Communication

The most flexible way to coordinate behavior is to explicitly communicate information between two agents. However many factors drive the military to minimize verbal communication (it may be difficult to transmit because of terrain and environmental factors, it increases the cognitive load on the agents that initiate and receive them, and it can be jammed, intercepted, or used to localize the position of an agent). Explicit communication is usually in natural language, and is one of the most timely types of communication.

In TacAir-Soar, explicit verbal communication is done via simulated radios (using the radio PDUs). There are a total of approximately twenty-five different message types that TacAir-Soar agents can send and receive (these cover the types of coordination covered in the previous section including messages for coordinating standard and tactical maneuvering, requesting and sending information about other planes, employing weapons, and changing the lead.

Communication with TacAir-Soar is natural enough so that it is possible for humans to fly in section with it using the HIP simulator interface [van Lent and Wray, 1994]. The HIP interface allows humans to fly either as lead or wingman (or even as an E-2) and compose messages that TacAir-Soar can understand, while receiving commands or acknowledgements from TacAir-Soar.

Coordination Capabilities

In this section, we draw together the capabilities required for coordination in the tactical air domain. This is based on the types of coordinated behavior (maneuvering, sensing, employing weapons, etc.), and the methods for sharing knowledge (doctrine and tactics, mission briefings, etc.). These capabilities serve as a requirements list for constructing an agent that can coordinate with others in domains such as tactical air combat. For each capability, we also describe how TacAir-Soar implements it.

Extensive Knowledge Base

Each agent must have an extensive knowledge base that includes all of the tactics and doctrine applicable to its possible roles in the missions in which it will participate. For example, a wingman must have the same knowledge of doctrine and tactics as the lead, so that the wingman can take over when necessary. Much of this knowledge is required even without coordination, but some will be unique to coordination activities, such as section-level tactics.

In TacAir-Soar, all of its knowledge is encoded in a rule-base of over 1400 rules. Its doctrine and tactics are encoded as a hierarchy of intertwined goals that are dynamically instantiated based on the current situation and mission.

Parameter-driven Behavior

An agent must be able to perform a variety of activities in coordination with others, such as defined by a mission briefing. The agent's behavior must be parameterized so that the knowledge relevant to the current mission is used. These may sound trivial, but for some complex missions, the information in the briefing may involve fragments of plans that the agent must integrate into its overall behavior at the appropriate times. Thus, the generators of the agent's behavior must be flexible enough so that they can be modified during a briefing.

Although one might be tempted selectively to build the knowledge base of an agent during the mission briefing, this would greatly restrict the abilities of that agent during the execution of a mission because of the dynamic nature of missions. For example, once the planes have taken off and are headed to their original CAP station, the situation may change so that they are redirected to a different CAP station.

In TacAir-Soar, all mission-related behavior is based on a representation of the current mission that is held in a working memory. This can be examined by the rules that make up its long-term knowledge. The mission can be specified at briefing time, but also can be dynamically changed during the mission.

Reactive Execution

In order to respond quickly to changes in a partner's behavior, an agent must be reactive. Of course, computer generated forces must in general be reactive, but coordination requires that they sometimes closely monitor the activities of other friendly agents. When flying in a section, the wingman must constantly monitor the actions of the lead, as well as the current spacing between the planes.

In TacAir-Soar, the wingman's main goal is to fly in formation with the lead. Rules are constantly monitoring the lead's actions and the position of the wingman relative to the lead. Whenever the wingman is out of position, rules fire to modify the heading, speed, or altitude as necessary.

Interruptible Processing

In being reactive, an agent is changing its behavior in response to the environment, however it is not performing any extensive reasoning, nor is it necessarily interrupting its ongoing goals. However, when an agent is communicating with other agents, it must often interrupt its current goals both to process the communication and to change its behavior in response to a message. For example, an agent may be flying an intercept of a bandit based on previous information from an E-2. When a new message arrives with new position information on the bandit, the agent must acknowledge the message, possibly abandon its current heading and compute a new heading.

In Soar, we have split the processing of incoming communications into two steps. The first is a high priority activity that categorizes the message and modifies the internal state of the agent in response to the message. The purpose is for this to happen quickly before other messages overwrite it. Following this, rules sensitive to the change will suggest changes to the current activities that the agent is pursuing. A more extensive examination of the problem of integrating communication (and natural language processing) within Soar systems has been done within the context of modeling the NASA Test Director, who is responsible for coordinating the launch of the Space Shuttle [Nelson *et al.*, 1994].

Translate internal information into messages

In order to communicate with other agents, an agent must be able to translate its internal information about its goals, its perception of the world, and its current actions into a form that can be understood by other agents. To do this right in general requires solving the natural language generation problem.

In the current version of TacAir-Soar, we are finessing the general problem and using an *ad hoc* approach where we prespecify the messages that the system can generate and when it should generate them. Thus, our agents do not explicitly plan their communications nor do they dynamically construct messages from the appropriate pieces. Instead they fill in prespecified templates. This approach has been successful for the limited types of communication our agents need to produce, but will break down when we get to more complex interactions and for these we are investigating more general approaches [Rubinoff and Lehman, 1994]. The form of our messages is based on the "Comm Brevity" lists of terms used by Navy pilots. This list contains over 150 terms, of which we use only those required for our current level of coordination, which is approximately 30 terms.

The most problematic type of communication is when an agent wishes to refer to another plane. For example, when the lead wishes to tell the wingman that they are committing to a bandit, the lead needs to specify which bandit it is. Internal to the lead, this may be represented by an internally generated name (such as B12), but the lead can not use that in the communication. Instead, the lead must use positional information, such as the bearing, range, and altitude of the bogey. This is problematic because the positional information is inexact and time dependent.

Translate messages into internal information

The converse of the prior problem is translating messages from other agents into an internal representation that the agent can work with. As above, to do this right in general requires solving the natural language understanding problem.

In the current version of TacAir-Soar, we are also finessing this problem by only accepting the message types that our agents generate (although we are also examining more general approaches [Rubinoff and Lehman, 1994]). By limiting the types of messages the system can accept, it is straightforward to translate the messages into the internal goals, actions, and state information of our agents. As above, the most problematic task is handling references to other agents, and this is done by finding the agent in the environment that most closely matches the description it is sent [Jones and Laird, 1994].

Conclusion

The purpose of this paper is to examine the capabilities required in a computer-generated force to support coordination. We have studied the low end of coordination as implemented in TacAir-Soar, where there are tight interactions between the agents involved. TacAir-Soar is proof that such coordination is possible, but that it required knowledge-rich, reactive, interruptible processing, with high frequency of relatively short messages. Our long term goal is to study coordination across the command hierarchy. As we move up the command hierarchy, we would expect that the frequency of the messages will decrease and the length of the message will increase, placing less emphasis on reactivity and interruptibility and more emphasis on the process of interpreting and generating messages.

Acknowledgements

This research was supported at the University of Michigan as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Research Laboratory. The authors would like to thank BMH Associates, Inc. for their technical help, and to the other members of the Soar/IFOR project.

Biographies

John E. Laird is an associate professor of Electrical Engineering and Computer Science and the director of the Artificial Intelligence Laboratory at the University of Michigan. He received his B.S. degree in Computer and Communication Sciences from the University of Michigan in 1975 and his M.S. and Ph.D. degrees in Computer Science from Carnegie Mellon University in 1978 and 1983, respectively. His interests are centered on creating integrated intelligent agents (using the Soar architecture), leading to research in problem solving, complex behavior representation, machine learning, cognitive modeling.

Randolph M. Jones received his Ph.D. in Information and Computer Science from the University of California, Irvine, in 1989. He is currently an assistant research scientist in the Artificial Intelligence Laboratory at the University of Michigan. His research interests lie in the areas of intelligent agents, problem solving, machine learning, and psychological modeling.

Paul E. Nielsen is an assistant research scientist at the Artificial Intelligence Laboratory of the University of Michigan. He received his Ph.D. from the University of Illinois in 1988. Prior to joining the University of Michigan he worked at the GE Corporate Research and Development Center. His research interests include intelligent agent modeling, qualitative physics, machine learning, and time constrained reasoning.

References

[Calder *et al.*, 1993] R. Calder, J. Smith, A. Courtenmanche, J. Mar, and A. Ceranowicz. ModSAF behavior simulation and control. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, 1993. Available as Technical Report IST-TR-93-07 from the University of Central Florida.

[Jones and Laird, 1994] R. M. Jones and J. E. Laird. Multiple information sources and multiple participants: Managing situational awareness in an autonomous agent. In *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, May 1994.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3), 1987.

[Nelson *et al.*, 1994] G. Nelson, J. F. Lehman, and B. E. John. Experiences in interruptible language processing. Unpublished, 1994.

[Powell and Hutchinson, 1993] D. R. Powell and J. L. Hutchinson. Eagle II: A prototype for multi-resolution combat modeling. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, 1993. Available as Technical Report IST-TR-93-07 from the University of Central Florida.

[Rosenbloom *et al.*, 1994] P. S. Rosenbloom, W. L. Johnson, R. M. Jones, F. Koss, J. E. Laird, J. F. Lehman, R. Rubinoff, K. B. Schamb, and M. Tambe. Intelligent automated agents for tactical air simulation: A progress report. In *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, May 1994.

[Rubinoff and Lehman, 1994] R. Rubinoff and J. F. Lehman. Natural language processing in an IFOR pilot. In *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, May 1994.

[Shoham and Tennenholtz, 1992] Y. Shoham and M. Tennenholtz. On the synthesis of useful social laws for artificial agents societies (preliminary report). In *Proceedings of AAAI-92*. Morgan Kaufmann, 1992.

[van Lent and Wray, 1994] M. van Lent and R. Wray. A very low cost system for direct human control of simulated vehicles. In *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, May 1994.

Natural Language Processing in an IFOR Pilot

Robert Rubinoff
Carnegie Mellon University
Pittsburgh, PA 15213
rubinoff@cs.cmu.edu

Jill Fain Lehman
Carnegie Mellon University
Pittsburgh, PA 15213
jef@cs.cmu.edu

Abstract

The creation of autonomous intelligent forces (IFORs) for both large-scale distributed simulations and small-scale, focussed training exercises creates unique challenges for natural language processing. An IFOR's role will often be to replace one or more individuals in an engagement, making the ability to communicate in natural language key to its performance as well its acceptance by other participants. In this paper, we describe the capabilities an IFOR needs to communicate appropriately and discuss how the NL-Soar language system provides these capabilities for TacAir-Soar, an IFOR agent for beyond-visual-range combat.

Introduction: IFORs and Communication

The creation of autonomous intelligent forces (IFORs) offers the possibility of running both large-scale distributed simulations and small-scale, focussed training exercises with lower manpower, cost, and logistical support requirements than previously possible. However, since an IFOR's role will often be to replace one or more individuals in an engagement, the ability to communicate in natural language can be a key aspect of its overall performance. An agent that cannot communicate at all is severely limited in the roles it can play. An agent that uses only a highly restricted subset of natural language may be easily detectable as a computer-generated foe, one that can be "gamed" without providing the actual training experience that is the point of the exercise. Further, an agent that is unlikely to comprehend the subset of language actually used by human participants puts an undue burden on those participants to communicate in a way that it can respond to, again changing the rules of the game. Finally, an agent that is rigid in its communicative ability may introduce a brittleness into the simulation (i.e. a tendency to fail in unexpected ways) that has nothing to do with imperfections in strategic or tactical knowledge.

Although the need to address the problem of natural language processing for IFORs is clear, the problem is complicated by the diverse ways in which NL can be called on to augment the functionality of the agent. For example, in building TacAir-Soar, a jet-fighter pilot IFOR for beyond-visual-range combat [RTLR93, RJJ+94], an NL capability is needed for basic interaction among pilot, wing, and air intercept control (AIC), as well as for descriptive explanation both during flight and in after-action review. We are adapting the NL-Soar language system [LLN91, Lew93] to provide that capability.[1]

[1]Here, we discuss our current work in basic interactive communication; see [Joh94] for details on explanation in after-action review.

There are three main characteristics of communication during air combat that present challenges for this research. The first challenge stems from the nature of the task itself: language processing occurs in real-time, as a single aspect of behavior in a constantly changing situation. Thus, in order to adequately simulate a human pilot, an IFOR must comprehend and generate language at roughly human rates. If it is too slow, it will be unable to keep up with both the linguistic and non-linguistic demands of the environment. If it is too fast it may commit to actions before co-ordinating its behavior with other sources of information (e.g. visual information from the radar).

The second challenge stems from the nature of the implementation: NL-Soar must be integrated into the structure of an independently-designed system. The organization of TacAir-Soar (and of IFORs in general) derives from the nature of the task(s) it performs; there is no a priori reason to expect this organization to be consistent with the assumptions underlying NL-Soar's design.

The third challenge stems from the particular nature of language in the domain. NL-Soar was originally designed to process complete grammatical sentences. The language in the tactical air combat domain differs from this both by including "ungrammatical" utterances such as sentence fragments and by containing many special purpose constructions (e.g. "roger" or the use of call-signs). In the rest of this paper, we explore the implications of these challenges in more detail and discuss how we are addressing them.

Real-time Communication

Communication in an IFOR must occur in real-time. This is not a statement about how fast the system must run, per se. Rather, it is a theoretical statement about how processing must occur within the system. Put simply, people can comprehend at rates of about 250 msec/word (they tend to generate language a bit more slowly). Although there is variability (some words take as little as 50 msec, others may take closer to 1000 msec), the point is that, in general, the amount of time is linear in the number of words in the utterance. A number of design constraints follow from this simple regularity [LLN99], e.g. construction of the meaning of the sentence must proceed incrementally, different knowledge sources (e.g. syntax, semantics, pragmatics) must be applied in an integrated rather than pipe-lined or multi-pass fashion. NL-Soar provides these properties [LLN91, Lew93]. Briefly, the system relies on Soar's notion of impasse to control the search through its linguistic knowledge sources, and then on Soar's learning mechanism to compile those disparate pieces of knowledge into an integrated form that can be applied directly (i.e. in constant time/word) in the future.

To make the nature of integration in NL-Soar more concrete, consider Figure 1, a graphical representation of a particular system that uses NL-Soar for comprehension and generation. Linguistic processes, like all processes in Soar, are cast as sequences of operators (small arrows) that transform states (boxes) until a goal state is achieved. The triangles in the picture represent problem spaces which are collections of operators.[2] The comprehension problem spaces contain operators that use input from the perceptual system to build syntactic and semantic structures on the state; the generation problem spaces contain operators that use semantic structures to produce syntactic structures and motor output. Note that the there is a special problem space, labelled Top, which is connected to the perceptual and motor sys-

[2]For more details on how Soar uses problem spaces, states and operators to organize its processing see [New90, LNR87].

tems. The Top space is the only problem space designated by the Soar architecture; all other problem spaces are provided by the system designer. The *dotted* lines in the figure represent Soar's impasses which arise automatically when there is a lack of knowledge available in the current problem space. When an impasse arises, processing continues in subspaces until the goal state in the subspace is reached. Thick banded arrows represent the resolution of an impasse, when chunks are formed. Chunks are new pieces of knowledge that are added to the system. They combine those conditions in the pre-impasse problem space that were used to reach the goal state in the subspace with the actions performed in the subspace to reach the goal state.

What does this mean for NL-Soar? As an example, consider the arrival of a new word into the Top state in some established context. Now assume that we have never seen the word in a similar context in the past. An impasse will arise and problem solving will continue in the Comprehension spaces until we reach the goal state in which we have defined the appropriate syntactic and semantic structures. When we return those structures to the Top state, chunks will be formed. In this case the chunks will propose operators directly in the Top state the next time this word is seen in a similar context. In other words, the next time, no impasse will occur; the problem solving that took place in the subspaces has been integrated into a small number of Top space operators that execute directly to build the relevant structures on the Top state.

A consequence of relying on Soar's learning mechanism is that achieving real-time language behavior requires training NL-Soar offline in advance. (Requiring NL-Soar to "learn while doing" would be equivalent to expecting the pilot to learn the domain language while flying the plane in battle.) When first loaded, TacAir-Soar/NL-Soar's lexical, syntactic, semantic, and discourse knowledge are all separate; it's as if the IFOR knew all the rules for how to communicate but had no experience using them. Off-line training allows NL-Soar to learn from experience in a non-real-time setting. This gives the system the time it needs to integrate its disparate knowledge sources into "chunks" that NL-Soar can apply in a single step. It is this highly compiled form of language knowledge that models an experienced pilot and provides real-time language behavior on-line.

Integrating Language with the Task

As mentioned above, NL-Soar was developed independently of TacAir-Soar. Indeed, as with many NL systems, NL-Soar was developed independently of the need to actually *do* anything non-linguistic. But, of course, most language, and certainly the communication between a pilot and AIC or wing, is generated and comprehended in service of some task that is, itself, essentially non-linguistic. As a result, NL-Soar must be adapted to seemlessly integrate the language capability with those non-linguistic capabilities in the agent, e.g. perception, planning, reasoning about the task. We have successfully done this on a smaller scale in NTD-Soar, a non-IFOR agent[3]. The structure of NTD-Soar, shown in Figure 1, is quite different from that of TacAir-Soar. In particular, NTD-Soar models switching between multiple tasks by invoking each task from the Top problem space; if a task is interrupted, its state is preserved in the Top space until it can be resumed. In essence, NTD-Soar models tasks in a fashion similar to co-routines. This structure allows language to be integrated easily by treating it as just another task. Information is transferred

[3]NTD-Soar is a model of the NASA test director who is responsible for coordinating many facets of the testing and preparation that the Space Shuttle must go through before it can be launched [NLJ94].

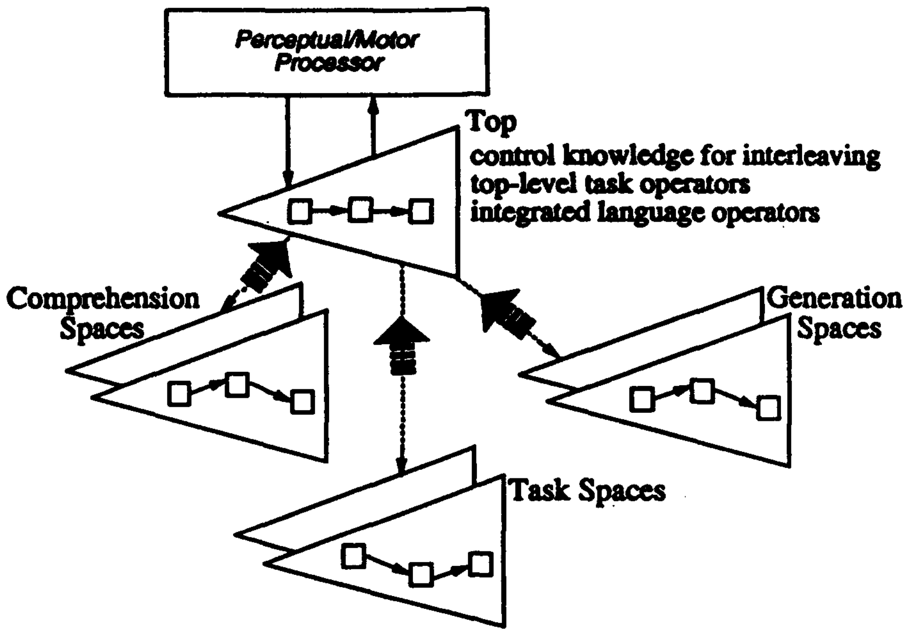

Perceptual/Motor Processor
Top
control knowledge for interleaving
top-level task operators
integrated language operators
Comprehension Spaces
Generation Spaces
Task Spaces

Figure 1: Structure of NTD-Soar

between language and other tasks by sharing the common Top state in the same problem space in which the task-switching control is done.

TacAir-Soar, in contrast, keeps only a single task active at a time, but it maintains a stack of levels of abstraction of that task, and each level stays active as long as it is being carried out. Thus TacAir-Soar uses Soar's top state to keep track of the "execute-mission" task, which stays active for the entire simulation. Under this will be a stack of sub-tasks, such as "mig-sweep", "intercept", "employ-weapons", and so on, each representing a more detailed view of what the agent is currently trying to do. Much of TacAir-Soar's knowledge of its current situation and goals is stored in sub-states associated with these sub-spaces, not on the top state. Thus if TacAir-Soar switched to language in its top state, as NTD-Soar does, it would lose much of this knowledge.

Because of the need to preserve TacAir-Soar's stack of subtasks, we have modified

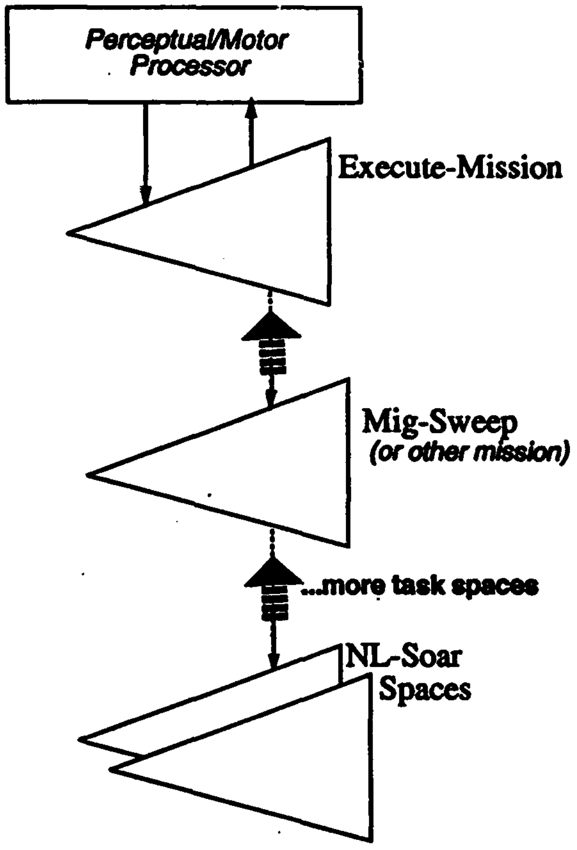


Figure 2: NL-Soar with Tac. ,oar

NL-Soar to operate at any level of the stack rather than just at the top. NL-Soar is thus invoked as a sub-task of the bottom-level task, preserving the stack. The resulting structure can be seen in Figure 2. This structure has the consequence of making language operate as a sub-task of the domain task(s), rather than as a separate task alongside them. While this is often reasonable, since the agent may be talking about what it's currently doing, it is not always so. Particularly in the case of comprehension, it may turn out that what someone says to the agent has to do with an entirely new task that the agent will start working on because of the communication. Given this and related problems, we are still exploring the overall issue of how best to integrate the structures of TacAir-Soar and NL-Soar.

Using Realistic Language

In addition to developing NL-Soar independently of TacAir-Soar, it was also developed independently of the language of the tactical air domain. This has two specific consequences. First, NL-Soar does not contain any of the domain-specific words and constructions used in tactical air combat. Furthermore, NL-Soar was designed to contain only *competence* knowledge. The competence-performance distinction [Cho65] reflects the difference between what people would recognize as fluent, grammatical speech, and actual speech as it occurs in everyday conversation. Thus NL-Soar must be able to comprehend and generate in accordance with domain-specific performance data, with all of its idiosyncratic constructions, ungrammaticalities, self-corrections, etc. In order to help adapt NL-Soar to this requirement, we have collected protocols of pilot/AIC and pilot/wing/AIC communication in a number of scenarios in a simulated environment. Doctrine with respect to communication is quite specific, stressing brevity and clarity. In addition to a highly specialized lexicon, this tends to result in a fairly agrammatical, telegraphic style, with periodic lapses into more standard English. This can be seen, for example, in Figure 3, which shows an excerpt from our protocols in which the AIC (whose call-sign is blue tail) guides the pilot (whose call-sign is dakota 204) to acquire his bogey (unidentified radar contact). (Punctuation has been added to aid the reader.) We can see here the use of domain-specific forms at all levels: syntactic (using call-signs in every sentence), semantic ("single" meaning "a plane flying unaccompanied"), and discourse ("roger" to acknowledge having heard someone). In addition, the pilot's last utterance demonstrates the kinds of "imperfect" speech (here pauses marked by "uh" and "eh") that NL-Soar must be able to comprehend and generate.

This challenge is easier to handle in generation than in comprehension, because NL-Soar has control over the structures that produce the surface form during generation; if it needs to generate an "ungrammatical" structure, it can simply build it and mark it as a special case. Of course, some care must still be taken to make sure that the special cases aren't too general (for example, the ability to say "roger" must not allow NL-Soar to utter any word as a single-word utterance). The problem is more complex during comprehension because the system is trying to recover the relevant structures from the surface form. Since NL-Soar can't know in advance whether the current utterance is following general English grammar, a domain-specific grammar, or represents some sort of speech error, the search space of possible interpretations can become quite large. There are a number of techniques that have been developed for dealing with this problem [FB86, Gra83, WB80, Leh90], although it remains for us to systematically evaluate the usefulness of each technique given the struc-

SPEAKER	START	END	UTTERANCE
AIC	13:30:41	13:30:46	dakota 2 0 4. blue tail. contact 2 7 0. approximately 50 miles.
Pilot	13:30:48	13:30:49	dakota 2 0 4 is clean.
AIC	13:30:52	13:31:00	roger. dakota 2 0 4 contact now 2 7 0. approximately 45 miles. appears to be single. contact at angels 18.
Pilot	13:31:01	13:31:04	dakota 2 0 4 roger. intermittent contact.
AIC	13:31:06	13:31:11	dakota 2 0 4. contact's now 2 7 0. approximately 35 miles.
Pilot	13:31:15	13:31:22	dakota 2 0 4. contact on the nose. uh bearing 2 5 5. eh 26 miles.
AIC	13:31:23	13:31:24	roger dakota that's your contact

Figure 3: Sample pilot conversation

ture of NL-Soar and the particular linguistic phenomena in the tactical air domain.

Summary

In this paper, we have discussed some of the challenges involved in providing an IFOR with communication capabilities that allow it to successfully simulate human behavior. These include the need to communicate in a real-time environment, to appropriately integrate the IFOR's language processing with its task operations, and the need to cope with domain-specific and ungrammatical language. Our adaptation of the NL-Soar language system to work with the TacAir-Soar IFOR agent continues to be guided by these concerns; despite a number of unresolved issues, we believe NL-Soar has the potential to provide TacAir-Soar with the necessary kinds of linguistic behavior.

Acknowledgement

This research was supported under subcontract to Carnegie Mellon University from the University of Michigan, as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). Tom Brandt, Bob Richards, Ken Smith, and Ed Harvey of BMH Inc. were instrumental in providing subject matter expertise.

References

[Cho65] N. Chomsky. *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA, 1965.

[FB86] P. K. Fink and A. W. Biermann. The correction of ill-formed input using history-based expectation with applications to speech understanding. *Computational Linguistics*, 12(1), 1986.

[Gra83] R. H. Granger. The nomad system: Expectation-based detection and correction of errors during understanding of syntactically and semantically ill-formed text. *American Journal of Computational Linguistics*, 9, 1983.

[Joh94] W. Lewis Johnson. Agents that explain their own actions. In *Proceedings of the Fourth Conference on Computer Generated*

Forces and Behavioral Representation, 1994.

[Leh90] Jill Fain Lehman. Adaptive parsing: A general method for learning idiosyncratic grammars. In *Proceedings of the Sixth International Conference on Machine Learnin g*, 1990.

[Lew93] R. L. Lewis. *An Architecturally-based Theory of Human Sentence Comprehension*. PhD thesis, Carnegie Mellon University, 1993.

[LLN99] J. Fain Lehman, R. Lewis, and A. Newell. NL-Soar: Architectural influences on language comprehension. In *Cognitive Architecture*. Ablex Press, 199? in press.

[LLN91] J. Fain Lehman, R. Lewis, and A. Newell. Integrating knowledge sources in language comprehension. In *Proceedings of the Thirteenth Annual Conferences of the Cognitive Science Society*, 1991.

[LNR87] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[New90] Allen Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.

[NLJ94] G. Nelson, J. F. Lehman, and B. E. John. Experiences in interruptible language processing. In *Proceedings of the 1994 AAAI Spring Symposium on Active NLP*, 1994.

[RJJ+94] Paul S. Rosenbloom, W. Lewis Johnson, Randy M. Jones, Frank Koss, John E. Laird, Jill Fain Lehman, Robert Rubinoff, Karl B. Schwamb, and Milind Tambe. Intelligent automated agents for tactical air simulation: a progress report. In *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, 1994.

[RTLR93] Jones R., M. Tambe, J. Laird, and P. Rosenbloom. Intelligent automated agents for flight training simulators. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, University of Central Florida, 1993. IST-TR-93-07.

[WB80] R. M. Weischedel and J. E. Black. Responding intelligently to unparsable inputs. *American Journal of Computational Linguisitics*, 6(2), 1980.

Biographies

Robert Rubinoff is a postdoctoral research fellow in Carnegie Mellon's School of Computer Science. He received his B.A., M.S.E., and Ph.D. from the University of Pennsylvania in 1982, 1986, and 1992, respectively; his dissertation research was on "Negotiation, Feedback, and Perspective within Natural Language Generation". His research interests include natural language processing, knowledge representation, and reasoning. He is currently working on natural language generation within the Soar project.

Jill Fain Lehman is a research computer scientist in Carnegie Mellon's School of Computer Science. She received her B.S. from Yale in 1981, and her M.S. and Ph.D. from

Carnegie Mellon in 1987 and 1989, respectively. Her research interests span the area of natural language processing: comprehension and generation, models of linguistic performance, and machine learning techniques for language acquisition. Her main project is NL-Soar, the natural language effort within the Soar project.

Working with ModSAF: Interfaces for Programs and Users

Karl B. Schwamb
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
schwamb@isi.edu

Frank Vincent Koss
Artificial Intelligence Laboratory
University of Michigan
1101 Beal Avenue
Ann Arbor, MI 28109-2110
koss@umich.edu

Dave Keirsey
Artificial Intelligence Center
Hughes Research Lab
Hughes Aircraft Company
Malibu, CA
keirsey@aic.hrl.hac.com

Abstract

In order to explore the domain of air-to-air combat with Soar, a unified theory of cognition used to model human behavior, it was necessary to interface Soar to vehicles which use the Distributed Interactive Simulation (DIS) protocol. Rather than create what would be in essence a simulator of fighter aircraft, the ModSAF simulation system was chosen to simulate fighter aircraft and provide a DIS interface. To link Soar and ModSAF, we have developed the Soar/ModSAF Interface (SMI). The SMI provides a simulated cockpit for Soar pilots. To guide others in the development of interfaces for other intelligent systems, this paper describes the SMI along with associated design constraints. Implementation details concerning functionality, modularity, and efficiency are addressed. We also identify issues arising from integration difficulties.

Introduction

Computer modeling of intelligent agent behavior is a concern to many researchers in the fields of cognitive science, artificial intelligence, and psychology. The Soar community is particularly interested in developing a model which encompasses a unified theory of cognition [Soar]. To this end, Soar researchers are interested in modeling agents that operate in challenging environments [TacAir]. Dynamic environments which require the application of a fair amount of domain knowledge offer a diverse set of problems that must be addressed in developing agents which simulate intelligent behavior. These problems require the development of a number of cognitive facilities in order to successfully simulate agent behavior and the unified approach of Soar is helpful in merging these facilities into a coherent whole.

One environment which provides these challenges is the domain of air-to-air combat. In this domain, fighter pilots must make quick decisions concerning enemy aircraft in the service of completing a mission. Developing an intelligent vehicle or robot to operate in such an environment is much too costly and the requirements of sensorimotor hardware development is too distracting from the central focus of behavior modeling. Given these concerns, the natural testbed for such development is a simulator. This simulator should provide a rich, high-fidelity world so that modeling of pilot behavior is not perverted by simulation artifacts. Fortunately, the ModSAF system [ModSAF] provides a rich simulation environment – it is designed to simulate vehicles in cooperation with conventional live force exercises.

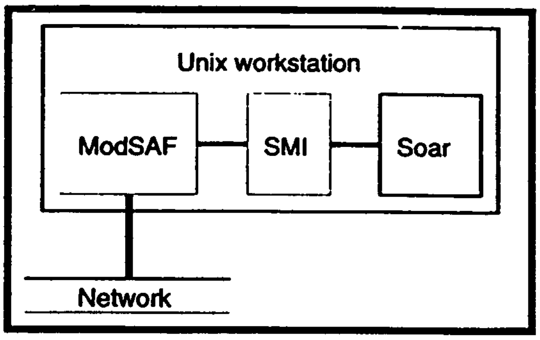


Figure 1: The Relationship of Soar, ModSAF, and the SMI.

ModSAF provides a platform for research into the control of all kinds of computer generated forces. In essence, ModSAF simulates the operation of DIS compatible vehicles. These vehicles can be directed by software-controlled agents or human beings. By using ModSAF, researchers can focus their work on the development of believable agents rather than on vehicle simulation issues, such as motion dynamics and DIS networking. Our work deals with the problems of interfacing artificially intelligent agents, modeled using the Soar system, to ModSAF. The module which supports the connection between the two systems is called the Soar/ModSAF Interface (SMI). Figure 1 shows the relationship between Soar, ModSAF, and the SMI.

```
Soar agent condor101> p o62
(O62 ^racetrack-dir O92
     ^racetrack-length O93
     ^type barcap
     ^risk-type high
     ^heading O46
     ^altitude O47
     ^speed O48
     ^id *none*
     ^e2c-id *none*
     ^level-of-experience low
     ^voice *none*
     ^ground-voice *none*)
(O92 ^value 0 ^units degrees)
(O93 ^value 36000 ^units meters)
(O46 ^value 0 ^units degrees)
(O47 ^value 7900 ^units feet)
(O48 ^value 320 ^units meters/second)

Soar agent condor101>
```

Figure 2: Working Memory Elements (WMEs) representing vehicle status information.

First we discuss the abstraction the SMI creates for Soar. We then move to the connection between Soar and ModSAF, and why other alternatives were not used. The "division of labor" among Soar, ModSAF, and the SMI is also explained, followed by implementation details.

The Cockpit Abstraction

Since Soar agents are constructed by modeling human pilots, it is imperative that the SMI provide an interface which emulates the environment of the human pilots - the aircraft cockpit. Soar agents receive input data corresponding to sensory information they would obtain from the cockpit environment, e.g. radar displays, radio messages, vehicle status indications, and visual sightings out of the cockpit canopy. This information is provided to Soar in the form of symbolic *working memory elements* (WMEs), not images or digitized audio. WMEs are the basic unit of information on which Soar acts. Soar agents also issue output commands to control the vehicle's motion, radar, weapons and radio. The specific Soar I/O WMEs defining the Application Programmer Interface (API) to the simulated cockpit are documented elsewhere[1]. An example of the WMEs representing the vehicle's status are shown in figure 2.

Unfortunately, there is no cockpit component provided by ModSAF. The SMI must create this facility for Soar agents via manipulation of the relevant ModSAF components and creation of new facilities. For this reason, the SMI is not a simple translation device between the two systems. Currently, control of a vehicle occurs through setting the desired state of the vehicle such as its speed, heading, and altitude. This level of control makes certain tasks difficult. For instance, it is not possible to cause the vehicle to climb without providing a desired altitude. Thus, attempting to stay in formation with another vehicle during a climb is very difficult since the agent must constantly monitor the altitude of the other vehicle and reset its desired altitude accordingly, rather than simply climbing until the other vehicle levels off. To more accurately model the vehicle control available to a pilot, the SMI will need to access lower-level ModSAF libraries which more closely correspond to cockpit controls.

The majority of the cockpit functionality is already provided by ModSAF in other libraries. Examples include the radar screen, missile launching, and detection of visual objects. These components are relatively straight-forward to access, requiring only the translation of units, reformatting, and reorganization of data. However, the missing cockpit components require the development of completely new functionality. A radar warning receiver and a radio device for inter-agent communication are examples. These additional components use ModSAF libraries at a low-level, if at all. Much more design and development is required for such enhancements.

Communication between Soar and ModSAF

The SMI design must be efficient and modular. Soar and ModSAF are designed as stand-alone systems and each system is designed to be the primary process running - not needing to function with other large processes. While these could be run as separate processes, Soar, ModSAF, and the SMI are incorporated into a single process to reduce communication overhead and increase overall system throughput. There is no need to encode and decode over a more general mechanism such as Unix sockets. This also enables high-bandwidth communication between Soar and ModSAF to be made more efficiently. Since both systems have a scheduler but one system must be in control of the primary scheduling, it was decided that ModSAF should call upon Soar at the appropriate times. This is natural since ModSAF controls the simulated "world" and Soar agents are agents in that world.

The incorporation of Soar, ModSAF, and the SMI into one process was fairly easy since all are written in the C language and utilize user-defined

[1]Users with access to the World Wide Web on the Internet can view this information using the URL http://krusty.eecs.umich.edu/ifor.

C libraries. The communication overhead is reduced by handling all I/O data flow through C function calls. Input to Soar systems takes the form of adding WMEs to Soar's memory. Output is carried out through placing WMEs in specific parts of the memory. To ease the task of adding Soar input working memory elements, an existing package was used that provides a convenient API to manage input working memory element retractions and assertions [SoarSIM].

For efficiency, the decision was made to only pass integer data values to Soar even though ModSAF calculated some data values using floating point numbers. The Soar input values were marked as being changed based on the rounded off values. This greatly reduced the number of memory updates needed during each Soar cycle. For small real number data items, the values were scaled into a larger integer range.

Until the advent of this project, Soar had been designed to support just one agent per process. Since Soar and ModSAF were to be run as a single process, the Soar system had to be modified to support multiple independent agents. Soar was generalized to allow the dynamic creation and destruction of agents, each operating with independent memories and I/O channels. There was no definitive critieria for defining an inter-agent communication mechanism, so none was created.

Functionality of Soar, ModSAF, and the SMI

Soar and ModSAF are very different systems. Each has certain capabilities that the other lacks because they were designed to different ends. When deciding where to implement certain functionality, in Soar, ModSAF, or the SMI, the strengths of the systems were the determining factors.

Ideally, ModSAF would be responsible for all vehicle and environment simulation and network interfacing, thus representing the aircraft and the world. Soar would be responsible for interpreting the world and controlling the plane, as a human pilot does. Such a clean separation is not possible. ModSAF provides a method for controlling vehicles called *tasks*. A number of tasks with different priorities can be assigned to a vehicle. The behavior of a vehicle is the result of the action of these tasks. Soar does not use these ModSAF tasks since a Soar agent typically deliberates about such things. The separation of vehicle simulation and tasks in ModSAF is not perfect, so the SMI fills in the gaps to provide a cockpit simulation to Soar agents. The Soar agent, for instance, sets the desired altitude and speed of the vehicle. This means that some of the functionality provided in tasks must be recreated in the SMI. This is due to the fact that there is no convenient way to use the functionality of the tasks without committing to use of more ModSAF machinery.

ModSAF has a library which provides facilities for editing various data structures. These graphical editors are used for such activities as creating vehicles and specifying missions to ModSAF vehicles. The Soar agents, however, use a representation of missions different from that provided by ModSAF. Therefore, the library that implements the editor and one that uses it were modified so that Soar-compatible missions can be created, saved, and modified. The modification of the ModSAF libraries had a number of advantages over writing an entirely new editor. First, the ModSAF editor has sub-modules defined for editing various data types, such as angles, speeds, and map locations. These sub-modules are used by the Soar mission editor. Second, modifying the editor library required much less time than would have been required to create a new editor from scratch. Even the time required to make these additional changes with each future releases of ModSAF is minor compared to the saved development time. Finally, as screen area is at a premium, reusing area that is already allocated benefits the user.

When Soar agents must communicate with one another, they must use some medium outside their I/O channels, just as humans do. In the air-to-air domain, inter-agent communication is carried out over radios. The generic radio interface of ModSAF[2] provides an implementation of this form of communication. Natural language character strings are sent in DIS Radio PDUs. Messages are generated by Soar as lists of WMEs (one per word) which are then turned into character strings by the SMI and passed to the ModSAF radio.

Soar and ModSAF do conflict in one area. ModSAF is a distributed simulation which causes problems when agents are created on separate hosts. When an agent is created, a user interface is created for that agent, whether it be a new X window or a new I/O stream interleaved onto standard input/output (used by the Soar Development Environment (SDE) [SDE]). This is no problem when one ModSAF is running on a local host. However, if more than one ModSAF is running and ModSAF's load balancing is active, then locally created agents will be simulated on remote hosts and their user interface will appear remotely. Fortunately, there are simple methods

[2]The generic radio library was added to ModSAF in version 1.0. Prior to this, interagent communication was performed using Message PDUs that were generated and interpreted by the SMI but sent and received by ModSAF

for forcing agents to remain on a local host. In the long term, it would be useful to find a method for allowing load balancing without interfering with the placement of the user interface. A more difficult issue in regards to load balancing is the moving of complex reasoning agents, such as the ones built in Soar. There is no simple mechanism to transfer both the complex reasoning state and the knowledge used in that reasoning to another machine, while the agent is interacting in the simulated world.

Implementation Details

The SMI must honor several design constraints. Although the primary focus is on the automated pilot which controls a single aircraft, there may be additional agents associated with a vehicle. A fighter aircraft may have a Radar Intercept Officer (RIO) and an Air Intercept Control (AIC) aircraft may have air controllers. Any of these agents may be created or destroyed at any point in the simulation; there is no preset scenario.

An arbitrary number of agents may exist in the Soar system and an arbitrary number of vehicles may exist in ModSAF. Not all of these vehicles may be controlled by Soar agents. Some agents may be controlled by other software modules or even by humans. The number of such entities is limited only by the processing speed and memory capacity of the host workstation. The SMI must be efficient so that the performance of Soar and ModSAF do not degrade due to excessive communication overhead between agents and their vehicles.

There are also implementation constraints on the SMI. Both Soar and ModSAF are designed as standalone systems and continue t[illegible] e ongoing development. The SMI must enable new versions of Soar and ModSAF to be incorporated. The Soar system is already implemented with a number of hook functions and configurable subsystems. Some of these facilities were generalized to work more effectively with external systems such as ModSAF, but no changes were needed to the Soar system releases. All SMI functionality is incorporated through Soar's extensible mechanisms. The SMI redefines Soar's scheduling command since ModSAF is in charge of scheduling, and adds a number of commands useful in the air combat domain. The SMI also adds a set of domain-specific right-hand side functions used in Soar productions.

ModSAF is also designed with modularity as an important goal. Hence, only one library out of over 100 was modified to incorporate Soar and the SMI. In this library, the SMI is implemented as a software layer connected to ModSAF at a level dealing with the aircraft vehicle simulation. The SMI calls upon a number of ModSAF libraries to help create the simulated cockpit. The ModSAF main program an[illegible] few additional libraries required minor addit[illegible]s to accommodate the SMI but their primary functionality was not altered.

ModSAF uses Motif and X for its graphical user interface (GUI) as well as standard input/output for its command line interpreter. Soar, which previously depended on standard input/output, was enhanced to include an X interface. This enabled Soar, ModSAF, and the SMI to present GUIs to the user while maintaining module independence. Each module opens a separate display connection to the user's console and receives a separate event stream. This design has the drawback that there is contention for screen real-estate due to a proliferation of separate windows.

The SMI GUI enables the user to control the simulation speed. This is helpful for speeding up the simulation in "dead spots" or slowing down the simulation to observe at a finer grain the changes in state. Additions to the SMI GUI are planned and will provide more dynamic control over ModSAF and the SMI. In addition to the SMI GUI, the ModSAF GUI was enhanced by adding two windows. The first provides orthographic projections of the PVD so that altitude relationships between vehicles may be depicted graphically. This window was augmented to provide some other desireable features missing from the ModSAF PVD: snail trails and radar volumes. Snail trails depict a history of vehicle positions over time by using a series of dots. The radar volumes are shown as fans indicating radar orientation, beam height, and beam width. The second window presents vehicle status information that is continually updated to clarify the status of vehicle position, orientation, radar sightings, and weapon employment.

An alternative interface to Soar was developed independently which utilizes standard input/output. This interface, the SDE, runs in Emacs. It removes the need for separate windows for Soar agents but forces the elimination of the ModSAF command line interpreter. Both Soar interfaces have their uses and Soar developers have not fully committed to one or the other.

Conclusion

The problem of connecting Soar to ModSAF has brought some interesting technical challenges. The challenges have helped the Soar system developers to generalize Soar's extension mechanisms enabling all Soar users to benefit. And the ModSAF environment has been an effective tool enabling Soar agent developers to focus more closely on modeling human pilots.

Acknowledgements

This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan, as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). Critical support has been provided by Dennis McBride of ARPA/ASTO; Tom Brandt, Bob Richards, and Ed Harvey of BMH Inc.; and Andy Ceranowicz and Joshua Smith of Loral Inc.

References

[ModSAF] R. Calder, J. Smith, A. Courtemanche, J. Mar, A. Ceranowicz (1993). ModSAF behavior simulation and control. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*. Univ. of Central FL, IST-TR-93-07.

[Soar] P. Rosenbloom, J. Laird, A. Newell (Eds.) (1993). *The Soar Papers: Research on Integrated Intelligence*. Cambridge, MA: MIT Press.

[TacAir] R. Jones, M. Tambe, J. Laird, P. Rosenbloom (1993). Intelligent automated agents for flight training simulators. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*. Univ. of Central FL, IST-TR-93-07.

[SDE] Michael Hucka (March 1994). The Soar Development Environment: User's Manual. Artificial Intelligence Laboratory, University of Michigan, unpublished manuscript.

[SoarSIM] Robert H. Guttman and Scott B. Huffman (June 1992). SoarSIM: A Soar Simulation-Building Tool (version 1.1 user's manual). Artificial Intelligence Laboratory, University of Michigan, unpublished manuscript.

Biographies

Karl B. Schwamb is a Senior Programmer Analyst on the Soar Intelligent FORces project at the University of Southern California's Information Sciences Institute. He is primarily responsible for the maintenance of the Soar/ModSAF interface software described in this article. He received his M.S. in Computer Science from George Washington University.

Frank Vincent Koss is a Systems Research Programmer in the Artificial Intelligence Laboratory at the University of Michigan, where he is developing the interface between the Soar architecture and the ModSAF simulator. He received his BS in computer engineering from Carnegie Mellon University in 1991 and his MSE in computer science and engineering from the University of Michigan in 1993. He is a member of IEEE and AAAI.

Dave Keirsey is a member of the senior staff, computer science in the Informaton Science Laboratory at Hughes Research Laboratories, where he is has been involved in the development of behavior-based methods for autonomous systems for both real and simulated environments. He received his PhD in computer science from University of California, Irvine in 1983.

Building Believable Agents for Simulation Environments: Extended Abstract

Milind Tambe[*], Randolph M. Jones[**], John E. Laird[**], Paul S. Rosenbloom[*], Karl Schwamb[*]

[*] Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
email: {tambe, rosenbloom, schwamb}@isi.edu

[**] Artificial Intelligence Laboratory
University of Michigan
120 ATL Building
1101 Beal Ave
Ann Arbor, MI 48109
email: {rjones,laird}@eecs.umich.edu

1. Introduction

The goal of our research effort is to develop generic technology for intelligent automated agents in simulation environments. These agents are to behave believably like humans in these environments. In this context, believability refers to the indistinguishability of these agents from humans, given the task being performed, its scope, and the allowable mode(s) of interaction during task performance. For instance, for a given simulation task, one allowable mode of interaction with an agent may be typewritten questions and answers on a limited subject matter. Alternatively, a different allowable mode of interaction for the same (or different) task may be speech rather than typewritten words. In all these cases, believability implies that the agent must be indistinguishable from a human, given the particular mode of interaction. Such an agent technology can potentially provide *virtual humans* for the multitude of *virtual reality* environments under construction. Its applications can be found in many fields, including entertainment [1], education [5, chapter 3], and training [2].

To begin this effort, we have focused on creating specific automated agents for simulated tactical air combat. The automated agents act as the *virtual pilots* for simulated aircraft, and will participate in exercises with real Navy pilots. These exercises will aid in training Navy pilots, development of tactics, and evaluation of proposed hardware. This is a non-trivial task, with many real-world complexities, and as such it offers several advantages. It pushes research based on real-world needs on topics such as reactivity, real-time reasoning, planning, episodic memory, agent modeling, temporal reasoning, explanation, and natural language understanding/generation. Furthermore, it forces the integration of all of these component AI technologies, because it requires a single automated agent to perform all of the functions performed by a pilot in air combat. Simultaneously, however, as a simulation task, it delimits the component technologies to be integrated. For example, it does not force the integration of vision or locomotion components. Finally, the task also imposes external metrics for success.

The task also poses an important constraint: the automated agents must believably act and react like trained human pilots. These agents are to take part in exercises with other human pilots. If human trainees identify our agents as automated pilots, they may take advantage of specific known characteristics of their behavior. Training in such a situation could actually be harmful. For instance, if the automated agents do not react as quickly as other human pilots (or react too quickly), trainees may learn to act too aggressively (or not aggressively enough) in a real aerial combat. Additionally, if the agents behave unrealistically, observers and tacticians at "ground control" (who can watch the simulated combat from different perspectives), may not be able to develop realistic tactics and strategies.

Thus, this task requires the development of believable automated pilots. For this fixed task, believability refers to the indistinguishability of the automated pilot from a human pilot, given the scope of the task, and the allowable modes of interaction. The scope of the task depends on (at least) the number of aircraft involved on each side, e.g., whether it is a one "friendly" aircraft versus one "enemy" aircraft (1v1) air-combat situation, or a 2v1, or 2vN situation. The allowable modes of interaction depend on whether it is a Beyond Visual Range (BVR) combat situation, where pilots only get radar information about the enemy aircraft, or Within Visual Range (WVR) combat situation, where the pilots can also directly see the enemy aircraft. In 2v1 (or 2vN) combat situations, additional modes of interaction are possible: the pilots of two or more "friendly" aircraft may communicate via radios, electronic data links, or even by executing simple maneuvers. A human observer at "ground control" adds even more modes of interaction. He/she can observe the combat in progress on a TV monitor, zoom in and out on it, focus on the maneuvers of a particular aircraft, and so on. A passive observer can only observe the combat in progress, while an active observer can supply the pilots new information or commands over the radio.

The specific scope of the task, together with the choice of certain modes of interaction, dictates the capabilities an agent must possess for believability. These capabilities define a certain *level of believability*. If the agent possesses these capabilities, then we refer to it as having (or being at) this level of believability. For instance, consider a 1v1 BVR air-combat situation, with no observers, and with a single human pilot engaged in combat with a single automated agent. The only mode of interaction is what the human pilot can view of the automated agent's actions on its radar. The capabilities required for believability are that these actions must appear like those of a trained human pilot. An agent with these capabilities has a certain (moderate) level of believability. Suppose we add a passive observer to this situation. Since the observer can watch the automated agent's actions much more closely, the agent must have a higher level of believability. As we add more aircraft, an active observer, and switch to WVR, the agent must have even higher levels of believability, with requirements for capabilities such as natural language (and speech) understanding/generation to support different types of radio communication.

The levels of believability provide us a means of staging an attack on this problem (and correspondingly staging the system development effort). Thus, to begin this effort, we have focused on an agent at a moderate level of believability: an agent for 1v1 BVR air-combat, with a passive observer. Even at this level, the task remains highly knowledge- and capability-intensive. Trained Navy pilots possess vast knowledge about different mission types, tactics and maneuvers, performance characteristics of the aircraft, radar modes, missile types and so on. The challenge for constructing an automated agent is then to integrate this knowledge into a single system, along with the following capabilities:

1. *The agent must be extremely flexible in its behavior:* Situations in air combat can change very rapidly. Unexpected events can occur, e.g., an on-target missile may fail to explode, or an aggressive adversary may engage in some preemptive action disrupting an ongoing maneuver. Accordingly, the agent must respond flexibly to the evolving situation.

2. *The agent must act/react in real-time:* Since a human may be interacting with the agent in real-time, the agent must act/react in real-time as well.

3. *The agent must try to interleave multiple high-level goals*: For this task, the agent must continuously attend to at least three high-level goals: (a) executing maneuvers to destroy the opponent; (b) surviving opponents' weapon firings; and (c) interpreting opponents' actions. Given the need for real-time response, the agent must be capable of rapidly switching among these goals (or achieving them in parallel).

4. *The agent must conform to human reaction times and other human limitations:* As

discussed earlier, the agent must not react to input data faster (or slower) than a human pilot would. The agent must also not maneuver the simulated aircraft like a "superhuman", e.g., it must not make very sharp turns. Finally, the agent must exhibit some unpredictability in its behavior, when appropriate.

5. *Others:* Some other capabilities such as planning, temporal reasoning, are also required for this task in limited proportions.

Note that, because a passive observer can watch an automated agent more closely than what is visible on radar, this additional level of believability requires more accurate modeling of human reaction time and physical limitations.

2. Developing Believable Pilot Agents

The basis of our work on developing automated agents is the Soar integrated architecture [4, 6] (Due to space constraints, we will assume that the reader has some familiarity with the Soar architecture). Some of the characteristics of this task are particularly well-suited for Soar. First, Soar is a single unified architecture for the research, development and integration of various component AI technologies. Second, Soar represents a developing unified theory of cognition, which is advantageous, given the constraint of psychological verisimilitude (e.g., limitation on reaction time) in this task.

The automated pilots for the 1v1 BVR air-combat task are based on TacAir-Soar, a system developed within the Soar architecture, which currently includes about 1100 productions. TacAir-Soar encodes the basic task knowledge for an agent in a set of problem spaces. A particular automated agent is realized by initializing TacAir-Soar with a specific set of parameters, such as its mission, the level of risk it can take for the mission, and the kind of weapons it has available.

The current design of TacAir-Soar is guided by two sets of constraints: the task requirements (as specified by the targeted level of believability), and the Soar architecture itself. Consider the key requirement of flexibility of behavior. This has turned out to be a strong constraint on the design of problem spaces and operators. For instance, any maneuver consisting of a sequence of actions is implemented not as a single monolithic plan, but rather as a sequence of appropriately conditioned operators in a problem-space. This allows TacAir-Soar to respond flexibly to an evolving situation, and not remain rigidly committed to a specific plan. Furthermore, this constraint discourages highly specific, narrowly focused problem spaces. For instance, a problem space devoted solely to employing one type of missile may not allow the system to switch quickly to employing a different type of missile, as the situation rapidly evolves. In contrast, a problem space that combines the operators for employing different types of missiles facilitates such actions.

TacAir-Soar's highly reactive behavior derives at least in part from Soar's ability to react at a number of different levels [3]. Specifically, Soar can respond to new inputs at three levels: (i) in a single production firing, (ii) in a single decision, which involves firing multiple productions, or (iii) in a problem-space, which involves executing multiple decisions. Thus, as the situation changes, Soar can respond very quickly within the time-span of a single production firing. If needed, it may also respond after much deliberation in a problem space. Additionally, Soar's efficient implementation technology plays a large role in allowing it to respond in real time.

In achieving multiple high-level goals, TacAir-Soar faces an interesting issue: as limited by the Soar architecture, it cannot construct multiple goal/problem-space hierarchies (in parallel) in service of the high-level goals. TacAir-Soar can and does construct a goal hierarchy in an attempt to achieve the high-level goal of destroying the opponent. For instance, to achieve the goal of destroying the opponent it creates a subgoal to "destroy-with-missile". To achieve destroy-with-missile, it generates subgoals to get into missile firing range, and so on. However, TacAir-Soar cannot construct goal-hierarchies for its remaining high-level goals — survival and interpretation of opponent actions — in parallel. To address this limitation, TacAir-Soar opportunistically installs operators for these high-level goals into its existing goal hierarchy (without eliminating the hierarchy). This avoids the overhead of rebuilding the goal hierarchy, while allowing it to switch attention among different types of goals rapidly. While this solution has allowed TacAir-Soar to exhibit reasonable performance so far, it does have some disadvantages. First, by not representing the different goal hierarchies explicitly, the solution does hinder TacAir-Soar's ability to reason about the interactions between multiple goals. Second, it is unclear if the scheme will generalize beyond the targeted level of believability. For instance, it is unclear if natural language understanding/generation will fit into this scheme. Alternative solutions are currently under investigation.

TacAir-Soar's ability to adhere to human reaction times is hindered by the artificiality of the interface to the simulation environment. In particular, TacAir-Soar does not spend time physically manipulating different instruments (e.g., turning a knob), or decoding actual instrument displays (e.g., decoding radar displays). As a result, TacAir-Soar tends to react faster than human pilots in some situations. Therefore, deliberate delays have been set up to slow down some of TacAir-Soar's responses. Similarly, TacAir-Soar's turning maneuvers have been constrained so as not to exceed human capability. As for unpredictability, much of it occurs "naturally" in TacAir-Soar. In particular, while two complex situations may appear very similar to a human observer, they may be quite different from TacAir-Soar's perspective, leading TacAir-Soar to two different actions. To add to this unpredictability, TacAir-Soar does random selection among operators that are considered to be equally appropriate in a given situation.

3. Current Status and Future Plans

Currently, even with approximately 1100 productions, the TacAir-Soar system continues to perform well within real-time constraints. Agents based on TacAir-Soar are fairly capable and robust within a narrow range of missions for 1v1 BVR combat. Recently, in a demonstration organized for Navy personnel, these agents were tested against (constrained) human pilots. The demonstration was a success in that the agents were able to function adequately at this targeted level of believability, i.e., they were able to react realistically to the humans pilots.

We are currently extending TacAir-Soar to deal with co-ordinated multi-aircraft air-combat simulations. Essentially, we are extending TacAir-Soar agents to higher levels of believability, and hence need integration of capabilities such as natural language understanding/generation. Thus, so far, for this task, the levels of believability appear to be useful as a means of staging development, as well as for measuring believability. Whether this usefulness will continue in the future, or for other tasks, remains to be seen.

References

1. Bates, J., Loyall, A. B., and Reilly, W. S. Integrating reactivity, goals and emotions in a broad agent. Tech. Rept. CMU-CS-92-142, School of Computer Science, Carnegie Mellon University, May, 1992.

2. Jones, R. M., Tambe, M., Laird, J. E., and Rosenbloom, P. Intelligent automated agents for flight training simulators. Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation, March, 1993.

3. Laird, J.E. and Rosenbloom, P.S. Integrating execution, planning, and learning in Soar for external environments. Proceedings of the National Conference on Artificial Intelligence, July, 1990.

4. Laird, J. E., Newell, A. and Rosenbloom, P. S. "Soar: An architecture for general intelligence". *Artificial Intelligence 33*, 1 (1987), 1-64.

5. Moravec, H. *Mind Children.* Harvard University Press, Cambridge, Massachusetts, 1990.

6. Rosenbloom, P. S., Laird, J. E., Newell, A., and McCarl, R. "A preliminary analysis of the Soar architecture as a basis for general intelligence". *Artificial Intelligence 47*, 1-3 (1991), 289-325.

Event Tracking in Complex Multi-agent Environments

Milind Tambe and Paul S. Rosenbloom
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
email: {tambe, rosenbloom}@isi.edu

Abstract

The Soar-IFOR project is aimed at developing intelligent automated pilots for simulated tactical air-combat. One key requirement for an automated pilot in this environment is *event tracking*: the ability to monitor or track events instigated by opponents, so as to respond to them appropriately. These events include the opponents' low level actions, which the automated pilot may directly observe, as well as opponents' high level plans and actions, which the automated pilot can not observe (but only infer). This paper analyzes the challenges that an automated pilots must face when tracking events in this environment. This analysis reveals some novel constraints on event tracking that arise from the dynamic multi-agent interactions in this environment. In previous work on event tracking, which is primarily based on single-agent environments, these constraints have not been addressed. This paper proposes one solution for event tracking that appears better suited for addressing these constraints. The solution is demonstrated via a simple re-implementation of an existing automated pilot agent for air-combat simulation[1].

1. Introduction

The Soar-IFOR project is aimed at developing intelligent automated pilots for simulated tactical air-combat environments [11, 17]. These automated pilots are intended to participate in large-scale exercises with a variety of human participants, including human fighter pilots. These exercises are to be used for training as well as for development of tactics. To participate in such exercises, the automated pilots must act in a realistic manner, i.e., like trained human pilots. Otherwise, both the training and tactics development in these environments will not be realistic.

To act in a realistic manner, an automated pilot must, among other things, be responsive to events in its environment — it must modify and adapt its own maneuvers in response to relevant events. These events may correspond to simple actions of other pilots, such as changes in heading or altitude, which the automated pilot may directly observe on its radar. Alternatively, these events may involve the execution of complex, high-level actions or plans of other pilots, which the automated pilot can not directly observe. For instance, one crucial event is an opponent's firing a missile at an automated pilot's aircraft, threatening its very survival. Yet, the automated pilot cannot directly see the missile until it is too late to evade it. Fortunately, the automated pilot can monitor the opponent's sequence of maneuvers, and infer the possibility of a missile firing based on them, as shown in Figure 1. The automated pilot is in the dark-shaded aircraft, and its opponent is in the light-shaded one.

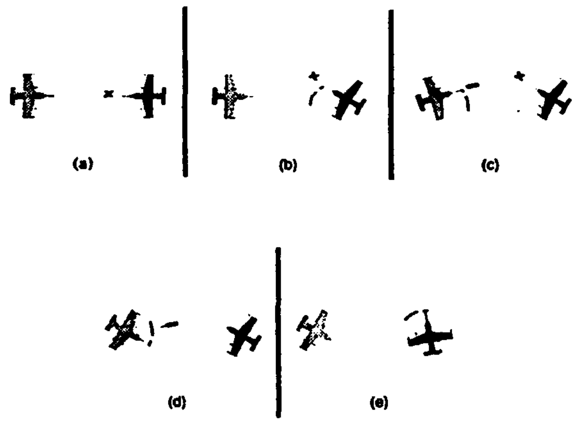


Figure 1: Manuevers of the automated pilot (in dark-shaded aircraft) and its opponent (in light-shaded one).

Suppose that initially the two aircraft are headed right toward each other as shown in Figure 1-a. The range (distance) between the two aircraft is

[1]This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan, as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). Critical support has been provided by Dennis McBride of ARPA/ASTO; and Tom Brandt, Bob Richards, and Ed Harvey of BMH Inc.

more than 10-15 miles, so they can only see each other on radar. This range is slightly short of the range from which the opponent can fire a radar-guided missile at the automated pilot's aircraft. However, the opponent is already well-positioned to fire this missile once its range is reached. In particular, given that the two aircraft are pointing right at each other, the opponent's aircraft is at *attack heading* (a point slightly in front of the automated agent's aircraft, as shown by a small x in the figure). At this juncture, the automated pilot turns its aircraft as shown in Figure 1-b. Given that the opponent wants to fire a missile, she turns her aircraft in response to re-orient it to attack heading (Figure 1-c). In this situation, she reaches her missile firing range, and fires a missile (shown by -). While the automated agent cannot observe this missile, based on the opponent's turn it can infer that the opponent may be attempting to achieve attack heading as part of her missile firing behavior. Unfortunately, at this point, it cannot be certain about the opponent's missile firing, at least not to an extent where trained fighter pilots would infer a missile firing. However, if the opponent subsequently engages in an *Fpole* maneuver then that considerably increases the likelihood of a missile firing (Figure 1-d). This maneuver involves a 25-50 degree turn away from the attack heading (it is executed after firing a missile to provide radar guidance to the missile, while reducing the closure between the two aircraft). While at this point the opponent's missile firing is still not an absolute certainty, its likelihood is high enough, so that trained fighter pilots assume the worst, and react as though a missile has actually been fired. The automated pilot reacts in a similar manner, by engaging in a missile-evasion maneuver. This involves turning the aircraft roughly perpendicular to the missile-flight (Figure 1-e), which causes the aircraft to "drop-off" (become invisible to) the opponent's radar. Deprived of radar guidance, the opponent's missile is rendered harmless.

The above example illustrates that an automated pilot needs to continually monitor a variety of events in its environment, such as the opponent's turns and her (inferred) missile-firing behavior, so as to react to them appropriately. We refer to this capability as *event tracking*. Here, an event may be considered as any coherent activity over an interval of time. An event is similar to a *process* in qualitative process theory [8], as something that acts through time to change the parameters of objects in a situation. This event may be a low-level action, such as an agent's Fpole turn, or it may be a high-level behavior, such as its missile-firing behavior, which consists of a sequence of such turns. The event may be internal to an agent, such as maintaining a goal or executing a plan, or external to it, such as executing an action. The event may be instigated by any of the agents in the environment, including the agent tracking the events, or by none of them (e.g., a lightning bolt). The event may be observed by an agent, perhaps on radar, or it may be unobserved, but inferred. *Tracking* any one of these events refers to recording it in memory and monitoring its progress as long as necessary to take appropriate action in response to it. Tracking an event also includes the ability to infer the occurrence of that event from other events.

Event tracking is closely related to the problem of plan recognition [12], the process of inferring an agent's plan based on observations of the agent's actions. The term event tracking is preferred in this investigation, since it also involves events other than plans, and since it is a continuous on-going activity. However, more important than the terminology, of course, is gaining a better understanding of the nature of this capability. In particular, does the realistic multi-agent setting of air-combat simulation reveal anything new about event tracking? Given the complexity of this domain, answering this question in its entirety is beyond the scope of this single investigation. However, this paper takes a first step by focusing on events relating to the actions and behaviors of one or two opponents as they confront the automated pilot. Section 2 illustrates that even within this restricted context, the air-combat domain brings forth some novel constraints on event tracking. Following this, Section 3 presents one approach that we have been investigating to address these constraints. The key idea in this solution is a basic shift in the agent's reasoning framework: from the usual agent-centric to world-centric. Finally, Section 4 presents a summary and issues for future work.

2. Event Tracking in Air-Combat Simulation

The primary constraint on event tracking in air-combat simulation arises from the fact that this is a dynamic environment, where agents continually interact. This continuous interaction implies that the agents cannot rigidly commit to performing a fixed sequence of actions. Instead, they need high behavioral flexibility and reactivity in order to achieve their goals. For instance, in Figure 1-c, the opponent has to re-orient herself to a new attack heading in response to the automated pilot's turn in

Figure 1-b. If the automated pilot had turned in the opposite direction, so would have the opponent. A more complex interaction occurs in Figure 1-e, where the automated pilot's missile evasion maneuver is a response to the opponent's overall maneuvers in Figures 1-c and 1-d, which are identified as part of her missile firing behavior.

These types of agent interactions extend well beyond situations involving just two aircraft. For instance, consider a situation where there are two opponents attacking the automated pilot's aircraft, as shown in Figure 2-a. Again, the automated pilot is in the dark-shaded aircraft, and the opponents are in the light-shaded aircraft. These opponents may either closely co-ordinate their attack or they may attack independently. One method of close co-ordination in the opponent's attack is shown in Figure 2-b. Here, the opponent closer to the automated pilot's aircraft (the *lead*) leads the attack, while the second opponent, marked with **x** (the *wingman*) just stays close to the lead, and follows her commands. Thus, as the lead turns to gain positional advantage, the wingman needs to turn in that direction as well, so as to fly in formation with the lead, all the while making sure that she does not get in between the lead and the automated pilot's aircraft. Another method of close co-ordination is shown in Figure 2-c. Here, the opponents execute a coordinated *pincer* maneuver — as the lead turns in one direction, the wingman turns in the opposite direction, so as to confuse the automated pilot and attack it from two sides. There are other possibilities of co-ordinating the attack as well. Of course, the opponents may not co-ordinate their attack. They may instead try to gain positional advantage in the combat independently of each other, and attack independently. In all these situations, all three aircraft continually influence each other's actions and behaviors in different ways. If other aircraft are involved in the combat — for instance, if the automated pilot is coordinating its attack with a friendly aircraft — then they also interact with the other aircraft involved in the combat.

This dynamic interaction among the agents leads to the primary constraint on event tracking in this domain: an agent must be able to track highly flexible and reactive behaviors of its opponent. In so doing, the agent must take the appropriate agent interaction into account. Without an understanding of this interaction, an opponent's action may lead to unuseful or even misleading interpretation. For instance, the opponent's turn in Figure 1-c needs to be tracked as a response to the automated pilot's own turn in Figure 1-b. Otherwise, that turn may appear meaningless. Similarly, as shown in Figure

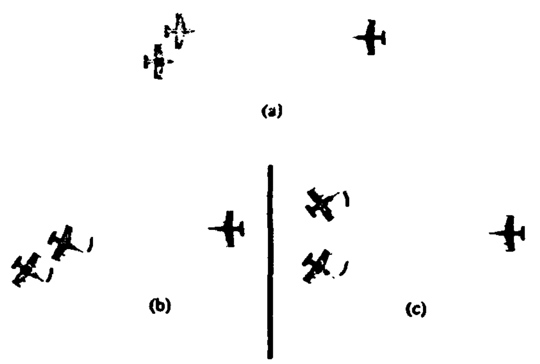


Figure 2: Agent interactions: (a) two opponents attacking the automated pilot's aircraft; (b) opponents stay close; (c) opponents stage a co-ordinated "pincer".

2, the wingman may mainly be reacting to its lead's turns, or she may be reacting to the automated pilot's aircraft independently. Understanding this interaction is important in tracking the wingman's actions.

A second related constraint here is that event-tracking must occur in real-time and must not hinder an agent from acting in real-time. For instance, in Figure 1, if the automated pilot does not track the missile firing event in real-time or does not react to it in real-time, the results could be fatal.

The third constraint on event tracking is that agents must be able to expect the occurrence of unseen, but on-going events. This constraint arises from the weakness of the sensors in this domain — an agent must sometimes track opponent's actions even though they are not visible on radar. For instance, suppose in the situation in Figure 2-c, the automated pilot concentrates its attack on the lead, and as a result the wingman (marked with **x**) drops off the automated pilot's radar. Here, given that the opponents are inferred to be executing a pincer maneuver, even though the wingman drops off the radar, some expectation about her position can be developed. Thus, the automated pilot can re-orient its radar and reset its mode to re-establish radar contact with the wingman if there is a need to do so later during the combat.

The fourth and final constraint on event tracking is that it is not a one-shot recognition task. Instead, it occurs on a continual basis, at least as long as it is relevant to the agent's achievement of its goals (such as the completion of its mission).

Thus, this domain poses a challenging combination of constraints for event tracking. The most novel constraint here is the first one. In previous investigations in the related areas of plan/situation recognition [12, 16, 6, 18, 3] — including one investigation focused on plan recognition in airborne tactical decision making [2] — this constraint has not been addressed. In particular, plan recognition models have not been applied in such dynamic, interactive multi-agent situations, and hence do not address strong interactions among agents and the resulting flexibility and reactivity in agent behaviors. In particular, these models assume that a single planning agent (or multiple independent planning agents) has some plans, and a recognizing agent recognizes these plans. The planning agent may be either actively cooperative (it intends for its plans to be recognized by the recognizing agent) or passive (it is unconcerned about its plans being recognized) [4]. The recognizing agent's job is to recognize these plans and possibly provide a helpful response. However, neither the recognizing agent, nor any other agents in the environment are assumed to have any influence on these plans. Consequently, these plan recognition models can rely on pre-compiled *plan libraries*, where each plan lists the sequence of events and the temporal relationships among the events [16]. However, such lists cannot be employed in tracking highly flexible and reactive agent behaviors. In particular, all possible variations on agent behaviors would need to be included in such lists, leading to a combinatorial explosion in the number of plans (unless a highly expressive plan language is developed).

Grosz and Sidner [9], in their work on discourse situations, attempt to partly address the above constraint on event tracking. They focus on what they characterize as the "master-slave" relationship between the planning agent and the recognizing agent assumed in plan-recognition models, and attempt to remedy it by using *shared plans*. Agents in their discourse situations arrive at a shared plan by establishing mutual beliefs and intentions about things such as their role in executing the plan. However, their discourse situations involve agents that are actively cooperative, while agents in air-combat simulation range from actively co-operative to passive to actively un-cooperative.

Interestingly, while plan-recognition systems have not dealt with such dynamic multi-agent situations, Distributed AI (DAI) systems, which have dealt with such situations, have not addressed the problem of plan recognition. There is some work in DAI on understanding other agents' plans [7]. However, it focuses on agents exchanging their plan data structures for active cooperation, rather than on plan recognition. Thus, the first constraint actually appears to give rise to a novel issue intersecting the areas of plan-recognition and DAI.

The remaining three constraints on event tracking — real-time performance, expectations and continuous tracking — have been addressed in previous research (e.g., in [6]). The next section presents an approach that we have been investigating for event tracking that addresses all four constraints outlined above.

3. Towards a Solution for Event Tracking

The key idea in the proposed solution for event tracking is based on the following observation. All of the agents in this environment possess similar types of knowledge, they have similar goals, and similar levels of flexibility and reactivity in their behaviors. In particular, an automated pilot agent that requires the capability to track events shares these similarities with its opponent. Thus, the key idea is that all the knowledge and implementation level mechanisms that the automated pilot agent uses in generating its own flexible behaviors may be used in service of tracking flexible behaviors of other agents.

To understand this idea in detail, it is first useful to understand how an agent generates its own flexible and reactive behaviors. Section 3.1 explains this by focusing on an automated pilot agent A_0 and its flexibility and reactivity. Section 3.2 then illustrates how A_0 may exploit this for tracking other agent's behaviors. Section 3.3 outlines the issues that arise in such an endeavor. Finally, Section 3.4 presents a simple re-implementation of an existing pilot agent based on the ideas presented in this section.

Note that while the solution presented here originated with the observation of similarity among agents, it is not necessarily limited to only those situations. For instance, it is possible that even though the other pilot agents may possess similar levels of flexibility and reactivity, they may be constrained in their behavior by their doctrine. To track these types of constrained behaviors, A_0 would need to use similar types of doctrine-based constraints in tracking behaviors of other agents.

3.1. An Agent's Own Behavior

This section illustrates how an automated pilot agent A_0 generates flexible and reactive behavior. This illustration is provided using a concrete implementation of A_0 in Soar [11, 17]. Soar is an integrated problem-solving and learning architecture that is already well-reported in the literature [14, 15]. The description below abstracts away from many of the details of this implementation, and mainly focuses on Soar's problem space model of problem-solving. Very briefly, a problem space consist of states and operators. An agent solves problems in a problem space by taking steps through the problem space to reach a goal. A step in a problem space usually involves applying an operator in the problem space to a state. This operator application changes the state. If the changes are what are expected from the operator application, then that operator application is terminated, and a new operator is applied. If the operator does not change the state, or if the changes it causes do not meet the expectations, then a subgoal is created. A new problem space is installed in the subgoal to attempt to achieve the expected effects of the operator. (Note that the system uses a procedural representation for these operator expectations — a declarative representation is not necessary. In particular, a procedural representation is sufficient to determine if the expectations are achieved.)

Figure 3 illustrates the problem spaces and operators A_0 employs while it is trying to get into position to fire a missile. In the figure, problem spaces are indicated with bold letters, and operators being applied in italics. In some problem spaces, alternative operators are also shown (these are not italicized). In the top-most problem space, named TOP-PS, A_0 is attempting to execute its mission by applying the *execute-mission* operator. This is the only operator it has in this problem space. The expected effect of this operator is the compl^tion of A_0's mission, which may be for example to protect its aircraft carrier. Since this expected effect is not yet achieved, a subgoal is generated to complete the application of *execute-mission*. This subgoal involves the EXECUTE-MISSION problem-space. There are various operators available in this problem space to execute A_0's mission, including *intercept* (to intercept an attacking opponent), *fly-racetrack* (to fly in a racetrack pattern searching for opponents when none is present), etc. In fact, in most of A_0's problem spaces there are always several such options available, and A_0 has to select a particular operator that would allow it to make the most progress. In this case, A_0 selects the *intercept* operator so as to intercept the opponent's aircraft. Given the presence of the opponent, this is the best option available.

A_0 attempts to apply the *intercept* operator. However, the expected effect of this operator — the opponent is either destroyed or chased away —

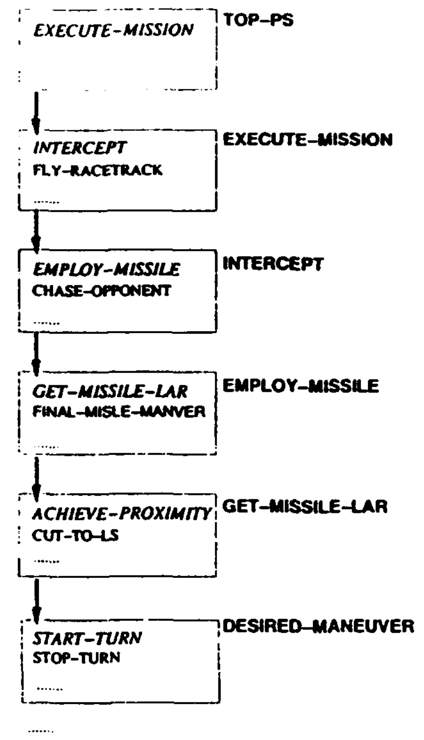


Figure 3: A_0's problem space/operator hierarchy. Boxes indicate problem spaces. Text in italics indicates currently active operator within a problem space.

is not directly achieved. This leads to a subgoal into the *intercept* problem space, where A_0 attempts to apply the *employ-missile* operator. However, the missile firing range and position is not yet reached. Therefore, A_0 subgoals into the EMPLOY-MISSILE problem space, and applies the *get-missile-lar* operator. (LAR stands for launch-acceptability-region, the position for A_0 to fire a missile at its opponent). The *get-missile-lar* operator results in the application of the *achieve-proximity* operator in a subgoal. Finally, this leads to a subgoal into the *start-turn* operator in the DESIRED-MANEUVER problem space. The application of this *start-turn* operator causes A_0 to turn. Another operator — *stop-turn* — will be applied to stop the aircraft's turn when it reaches a particular heading (called collision-course). This heading will be maintained until missile firing position is reached. At that time, the expected effect of A_0's *get-missile-lar* operator will be achieved, and hence it will be terminated. A_0 can then apply the *final-missile-maneuver* operator from the EMPLOY-WEAPONS problem space. The *final-missile-maneuver* operator may lead to subgoals in other problem spaces, not shown in the figure.

Thus, by subgoaling from one operator into another a whole operator/problem-space hierarchy is generated. The state in each of these problem spaces consists of a global portion shared by all of the problem spaces and a local portion that is local to that particular problem space. This organization supports reactive and flexible behaviors given appropriate pre-conditions (or conditions) for the operators, and the appropriate operator selection and termination mechanisms, as outlined in [13]. In particular, if the global state changes so that the expected effects of any of the operators in the operator hierarchy is achieved, then that operator can be terminated. All of the subgoals generated due to that operator are automatically deleted. Note that A_0 may also terminate an operator even if its expected effects are not achieved. This may be achieved if another operator is found to be more appropriate for the changed situation. For instance, suppose the opponent suddenly abandons the combat and turns to return to it base while A_0 is attempting to fire a missile at the opponent as shown above. In this case, the *chase-opponent* operator may be more appropriate than the *employ-missile* operator in the *intercept* problem space. Hence, A_0 terminates the *employ-missile* operator (all its subgoals get eliminated as well), and instead, A_0 applies the *chase-opponent* operator.

Since all of the above operators are used in generation of A_0's own actions, they will be henceforth denoted using the subscript *own*. For instance, *employ-missile*$_{own}$ will denote the operator A_0 uses in employing a missile. Operator$_{own}$ will be used to denote a generic operator that A_0 uses to generate its own actions. The global state in these problem spaces will be denoted by state$_{own}$. Problem-spaces that consist of state$_{own}$ and operator$_{own}$ will be referred to as *self-centered* problem spaces. The motivation for using this method for denoting states operators and problem spaces will become clearer below.

3.2. Tracking Other Agent's Behaviors

Given the similarities between A_0 and its opponent, the key idea in our approach to event tracking is to use A_0's problem space and operator hierarchy to track opponent's behaviors. We will first illustrate this idea in some detail using some simplifying assumptions. The detailed issues involved in operationalizing this idea will be discussed in Section 3.3.

To begin with, let us assume that A_0 and its opponent are exactly identical in terms of the knowledge they have of this domain, and all their other characteristics related to this domain. That is, A_0 and its opponent have identical problem spaces and operators at their disposal to engage in the air-combat simulation task. This simplifies A_0's event tracking task, since it can essentially use a copy of its own problem-spaces and operators to track the opponent's actions and behaviors. Operators in these problem spaces represent A_0's model of its opponent's operators. These operators are denoted using the subscript *opponent*. Thus, the *execute-mission* operator used in modeling an opponent's execution of her mission is denoted by *execute-mission*$_{opponent}$. Similarly, operator$_{opponent}$ will be used to denote a generic operator used by the opponent.

The global state in these problem-spaces represents A_0's model of the state of its opponent, and is denoted by state$_{opponent}$. Generating state$_{opponent}$ requires A_0 to model features such as the opponent's sensor input. Based on information such as the range of opponent's sensors, at least a portion of this state can be generated. However, other portions of state$_{opponent}$ may require fairly complex computation, essentially mirroring the computation that A_0 requires to generate all of the information in state$_{own}$. For instance, one important piece of information that is computed in state$_{own}$ is the "angle off" (the angle between the A_0's flight path and opponent's position). Mirroring this computation in state$_{opponent}$ will mean the computation of this "angle off" from the opponent's perspective (the angle between the opponent's flight path and A_0's position). For now, we make another simplifying assumption — that A_0 generates a detailed and accurate state$_{opponent}$ — and revisit this issue in Section 3.3.

The problem spaces consisting of state$_{opponent}$ and operator$_{opponent}$ discussed above are referred to as *opponent-centered* problem spaces. With the opponent-centered problem spaces, A_0 can essentially pretend to be the opponent. A_0 then tracks opponent's behaviors and actions by pretending to engage in the same behaviors and actions as the opponent. In particular, A_0 applies operator$_{opponent}$ to state$_{opponent}$, thus modeling the opponent's actual application of her operator to her actual state. Since A_0 is modeling the opponent's action, operator$_{opponent}$ does not change state$_{opponent}$. Instead, if the opponent takes some action in the real-world, then that change is modeled as a change in state$_{opponent}$. If this change matches the expected effects of operator$_{opponent}$, then that effectively corroborates A_0's modeling of

$operator_{opponent}$. (Note that as with A_0's $operator_{own}$, these expectations of $operator_{opponent}$ may also only be represented procedurally. This procedural representation is sufficient to match the expectations.) If these expectations are successfully matched, $operator_{opponent}$ is then terminated. As an example, consider *start-turn*$_{opponent}$ being applied to $state_{opponent}$. If the opponent actually starts turning, then the operator *start-turn*$_{opponent}$ is corroborated and terminated. Of course, low-level operators such as *start-turn*$_{opponent}$ are easy to corroborate in this manner, since the actions they model are directly observable. Others, however, may not generate low-level actions that are directly observable. One category of such operators are the higher level operators like *employ-missile*$_{opponent}$, which consists of a number of low-level actions. This issue will be discussed below.

This technique of event tracking, where an agent models another by pretending to be in that agent's position, has been previously used in automated tutoring systems [1, 19]. These tutoring systems need the ability to model the actions of the students being tutored. For this, these systems use student-centered problem spaces where states and operators model the students under scrutiny. This technique of modeling the student is referred to as *model tracing*. The approach proposed here for event tracking is thus based on this model tracing work. However, there are some significant differences. For instance, previous work has primarily focused on static, single-agent environments, where the agent being modeled is the only one causing changes in the environment [10]. There are some other differences as well. However, before exploring the impact of these differences, it is useful to first understand in detail how A_0 can perform event tracking using its opponent-centered problem spaces. This is explained below using the example from Figure 1. While this explanation does not directly describe the operation of an actual implementation, it is based on an actual implementation that will be described in Section 3.4. Basically, the description presented here will be used to motivate some representational modification leading up to the implementation described in Section 3.4.

Consider the situation in Figure 1-a. In this case, A_0 models the opponent's operator hierarchy as shown in Figure 4-a. A_0 is seen to accurately model this goal hierarchy, and in particular without any ambiguity about what actions the opponent is exactly engaged in. This is again a simplifying assumption, and we will return to it in Section 3.3. Figure 4-b shows A_0's own operator hierarchy corresponding to the situation in Figure 1-a. We assume that A_0 dovetails the execution of these operator hierarchies, communicating important relevant information from one to the other.

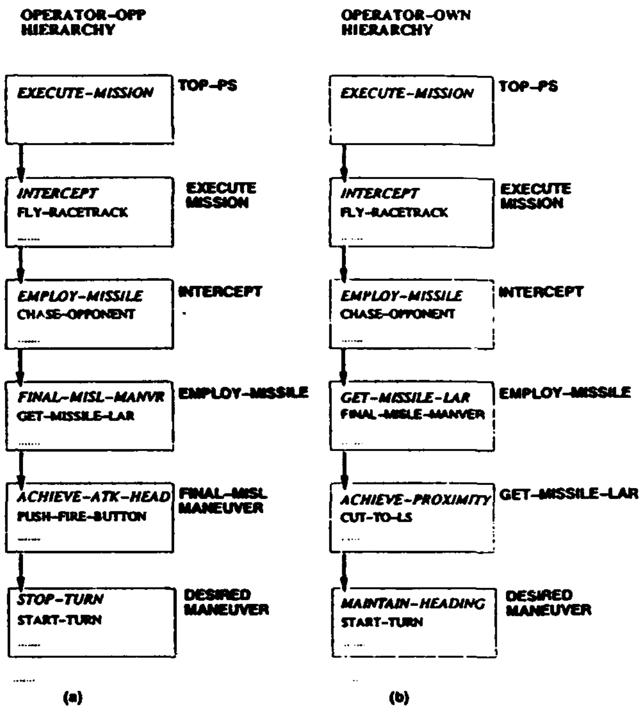


Figure 4: (a) A model of opponent's operator hierarchy, and (b) A_0's own operator hierarchy.

Consider the model of the opponent's operator hierarchy from Figure 4-a. One of the operators in this hierarchy is *final-missile-maneuvers*$_{opponent}$, which models the opponent's final missile-launching behavior. This is a high-level operator, and its expectations cannot be directly corroborated by observation. This operator is seen to generate a subgoal, where the first operator is *achieve-attack-heading*$_{opponent}$. This would require a *start-turn*$_{opponent}$ operator to turn to attack-heading. In Figure 1-a, attack heading is achieved, and $state_{opponent}$ encodes that fact. Hence, *stop-turn*$_{opponent}$ is being modeled as the current operator, to model the opponent's stopping her turn at attack-heading.

Now consider A_0's own operator hierarchy in Figure 4-b. A_0 is attempting to get into position to fire its own missile using the *achieve-proximity*$_{own}$ operator in the GET-MISSILE-LAR problem space. When the situation changes from Figure 1-a to Figure 1-b, A_0 selects the *cut-to-ls*$_{own}$ operator in place of the *achieve-proximity*$_{own}$ operator in the GET-MISSILE-LAR problem space. This operator is intended to increase the lateral separation between the two aircraft.[2] The *cut-to-ls*$_{own}$ operator causes A_o to turn its own aircraft as shown in Figure 1-b. As the aircraft turns to a particular heading, this new heading is modeled in state$_{own}$. Thus the *cut-to-ls*$_{own}$ operator leads to indirect modification of state$_{own}$.

This change in state$_{own}$ has to be communicated to state$_{opponent}$, to update A_o's heading in state$_{opponent}$. This leads to further modification in state$_{opponent}$, indicating that the opponent's attack heading is no longer achieved. Based on this modification, *achieve-attack-heading*$_{opponent}$ is re-activated (or re-applied). This operator again subgoals into the DESIRED-MANUEVER problem space where the *start-turn*$_{opponent}$ operator is reapplied. When the opponent starts turning, this operator is corroborated and terminated. The next operator in this problem space is *stop-turning*$_{opponent}$. When the opponent actually stops turning after reaching attack heading, as shown in Figure 1-c, state$_{opponent}$ is modified to indicate that opponent's attack-heading is achieved, and hence *stop-turning*$_{opponent}$ operator is corroborated. The change in heading in state$_{opponent}$ needs to be communicated back to state$_{own}$, so that A_o may readjust its heading in *cut-to-ls*$_{own}$ if required.

Continuing with Figure 1-c, the opponent's achievement of attack-heading also corroborates the *achieve-attack-heading*$_{opponent}$ operator, which is now terminated. A new operator from the FINAL-MISSILE-MANEUVERS problem space — *push-fire-button*$_{opponent}$ — is now applied. This operator predicts a missile firing, but it is known that that cannot be observed. Hence, *push-fire-button*$_{opponent}$ is terminated even though there is no direct observation to support that termination. However, the resulting missile firing is marked as not being highly likely. Nonetheless, this missile launch, even with its low likelihood, is communicated to state$_{own}$, so that A_o may react to it (for instance if A_o's mission forbids it from taking any risks at all). At this point, given the termination of the *push-fire-button*$_{opponent}$ operator, opponent's *final-missile-maneuvers*$_{opponent}$ operators is corroborated and terminated. Following that, an Fpole$_{opponent}$ operator in the EMPLOY-MISSILE problem space predicts an Fpole turn. This again generates a subgoal, back into the DESIRED-MANEUVER problem space and the *start-turn*$_{opponent}$ operator is reapplied. When the opponent executes her Fpole turn in Figure 1-d, the Fpole$_{opponent}$ operator is corroborated and terminated. At this point, all of the expectations for the high-level employ-missile$_{opponent}$ operator are corroborated; and hence state$_{opponent}$ is modified to indicate that a missile launch is highly likely. These changes in state$_{opponent}$ — the change in the opponent's heading and the highly likely status of the missile launch — are once again communicated to state$_{own}$. Based on the high likelihood of the missile launch, A_o activates the operator missile-evasion$_{own}$ to evade the incoming missile (Figure 1-e). This change in A_o's heading is once again communicated back to state$_{opponent}$.

Thus, A_o executes its own operators, and tracks opponent's actions and behaviors using the operator$_{opponent}$ and state$_{opponent}$. This can help A_o to track its opponent's behaviors, and address all of the constraints on event tracking outlined in *Section* 2. However, there are some important issues involved in addressing our earlier constraints with this approach. There are also some simplifying assumptions that we made in illustrating event tracking: (i) A_o and its opponent are identical; (ii) A_o performs all of the complex computation that is necessary to accurately model opponent's state; and (iii) A_o can accurately model opponent's operator hierarchy without any ambiguity. Relaxing these assumptions leads to some additional issues, which also relate to the constraints on event tracking. These issues are all discussed in the next Section.

3.3. Addressing Constraints on Event Tracking

The first constraint on event tracking was for an agent to track highly flexible and reactive behaviors of its opponent, while taking appropriate agent interactions into account. The use of opponent-centered problem spaces with operator$_{opponent}$ and state$_{opponent}$ helps in partly addressing this constraint (this was the motivation behind this approach to begin with). In particular, operator$_{opponent}$ can be activated and terminated in the same flexible manner as operator$_{own}$. There is complete uniformity in the treatment of the two

[2]Lateral separation is defined as the perpendicular distance between the line of flight of A_o's aircraft and the position of its opponent. When the two aircraft are pointing right at each other as in Figure 1-a, there is no lateral separation between the two aircraft. Increasing lateral separation provides a positional advantage.

types of operators.

However, these opponent-centered problem spaces by themselves do not address the issue of modeling the interactions among the different agents. In particular, the method outlined in Section 3.2 requires building one operator hierarchy for A_0, and one for each opponent, with their own global states. This leads to a situation where multiple compartmentalized operator hierarchies with their own global states are generated. Modeling the strong agent interactions present in this domain requires passing messages from one compartment to another. For instance, as described above, when A_0 changes heading, that information needs to be propagated from $\text{state}_{\text{own}}$ to $\text{state}_{\text{opponent}}$. Similarly, when the opponent fires a missile that information has to be communicated to $\text{state}_{\text{own}}$ from $\text{state}_{\text{opponent}}$. Similarly, if A_0 is to take some action depending on whether the *intercept*$_{\text{opponent}}$ operator is being executed, then that information would need to be propagated to A_0's compartment.

Given the level of interactions among A_0 and its opponents, this message passing can be a substantial overhead. Furthermore, there can be many aircraft involved in the combat, leading to an increase in the message passing overheads. This is particularly problematical given the second constraint on event tracking (of real-time performance) and the fourth constraint (which implies continuous agent interactions). Additionally, the communication among the different compartments essentially duplicates the information of one compartment in another. For instance, when a missile is fired, this information is duplicated in different compartments. Such duplication is problematical in terms of maintaining its consistency. If a missile is removed from one compartment, it must be removed from all of the others.

The solution we are investigating to alleviate the problem with this compartmentalization is to merge the different operator hierarchies for the different agents into a single compartment, which we will refer to as world-centered problem space (WCPS for short). WCPS eliminates the boundaries between different self-centered and opponent-centered problem spaces. Instead, the different operator hierarchies are maintained within the context of a single WCPS. There is also a single world state. This state includes A_0's own problem-solving state ($\text{state}_{\text{own}}$), A_0's model of the state of its opponent ($\text{state}_{\text{opponent}}$), as well as A_0's model of the states of other entities, including other opponents or friendlies in the world.

WCPS eliminates the need for passing messages to model interactions. Instead, interactions get modeled in terms of changes to the single global state. $\text{Operator}_{\text{own}}$ and $\text{operator}_{\text{opponent}}$ are directly able to reference this global state as well as other operators. Furthermore, the problem of duplication of information is avoided. For instance, a missile fired by the opponent gets modeled within this single global state as a single missile. Operator hierarchies modeling all of the different agents can directly react to this missile.

An additional benefit of the single global state in WCPS also relates to one of the assumptions mentioned in Section 3.2. In particular, A_0 need not perform all of the complex computation required in modeling opponent's state, but instead it may "re-use" some of the computation. Consider the example of the computation of "angle off" from the opponent's perspective, as mentioned in Section 3.2. With the global state in WCPS, A_0 does not need to recompute this "angle off". Instead, this is automatically computed in A_0's $\text{state}_{\text{own}}$, and this can simply be reused. In particular, A_0's $\text{state}_{\text{own}}$ already maintains the computation of "target aspect" from its own perspective (the angle between the opponent's flight path and A_0's position). This is precisely the definition of "angle off" the opponent's perspective. Thus, instead of computing the "angle off" from the opponent's perspective and "target aspect" from A_0's perspective separately, a single computation can be performed and used for both purposes. Of course, not all of the complex computation involved in generating the opponent's state can be avoided in this manner. The interesting research question then is determining what portion can be re-used in this manner, and how much extra computation is really necessary.

This shift from small self-centered and opponent-centered problem-spaces to WCPS related to the *objective* framework used in simulation and analysis of DAI systems [5], which describes the essential, "real" situation in the world. However, the focus of our work is on an individual agent using its world-centered model for event-tracking. While this model introduces a shift towards an objective point of view, by definition, it is an agent's subjective view of its environment. and may contain approximations in $\text{operator}_{\text{opponent}}$ and $\text{state}_{\text{opponent}}$.[3]

[3]Note that if the agents do not interact, then a single WCPS may not be appropriate, and separate problem spaces may be the right choice for modeling them.

The second constraint on event tracking relates to A_0's ability to track events in real-time. The key impact of this decision is on generating an accurate and unambiguous operator$_{opponent}$ hierarchy — one of the assumptions made in the previous section. In particular, this constrains the methods A_0 can employ in attempting to generate an accurate and unambiguous operator hierarchy. For instance, Ward [19] presents one general method for generating an unambiguous operator hierarchy. This method involves an exhaustive search over all possible operator applications until the one that creates the right expectations, i.e., one that matches the opponent's current actions, is created. If there is more than one such operator application, then one is chosen randomly. A wrong choice can be made in such situations. However, as soon as that is discovered, another exhaustive search can be performed. Given the real-time constraint on event tracking, this type of exhaustive search strategy can not be applied. While Ward suggests some heuristics to constrain the search, this remains a difficult problem. The WCPS approach at least provides a partial answer here. In particular, given the uniformity among operator$_{own}$ and operator$_{opponent}$ in WCPS, the mechanism employed in resolving ambiguity in operator$_{own}$ operators — search control rules — can also be used in resolving ambiguity in operator$_{opponent}$. Besides search control rules, another possibility for resolving ambiguity in WCPS is to generate the goal hierarchy bottom-up rather than top-down. While both of these are powerful tools in WCPS, their advantages and disadvantages in this context are not yet well understood.

The real-time constraint also raises the issue of abstractions in event tracking. In particular, Hill and Johnson [10] have recently argued that tracking an individual agent's actions in detail in a dynamic environment may prove computationally intractable. They advocate detailed tracking only where necessary, and reliance on abstractions elsewhere. In WCPS, abstractions in modeling an operator would imply that detailed subgoals for modeling that operator need not be generated. For instance, A_0 may not model the detailed operators used in accomplishing *get-missile-lar*$_{opponent}$. Thus, when *get-missile-lar*$_{opponent}$ is activated, it may not lead to any subgoals. However, when the opponent actually reaches the LAR (missile firing position), *get-missile-lar*$_{opponent}$ can be considered as corroborated and terminated. Unfortunately, this method of abstract modeling may not be appropriate for corroborating an operator such as *employ-missile*$_{opponent}$, which involves multiple maneuvers. In this case, the intermediate headings of opponent's aircraft may be important and just testing the terminating position may be an inappropriate test for corroboration. Automatic generation of the right levels of abstraction is an interesting issue for future work.

The third constraint on event tracking was the generation of expectations for an unseen, but on-going event. In WCPS, the application of an operator$_{opponent}$ in essence is the expectation for the opponent to execute a certain plan or action. Thus, this constraint can be addressed in a straightforward manner. However, since the event is unseen, there can be no corroboration of it. One possibility to deal with this situation is to terminate operator$_{opponent}$ if the relevant action is known to be unobservable (for instance, since the opponent's aircraft is not observable on radar).

The fourth constraint is related to the continuous nature of event tracking. The main implication of this constraint is the continuous interaction among agents, which as discussed above, leads to the move towards WCPS.

There were also three assumptions made in the previous section to simplify event tracking. The second and the third assumption, related to modeling of the opponent's state and operator hierarchy have been discussed above. However, the first one of the assumptions has not been discussed. This assumption is that the automated pilot agent A_0 and its opponent are identical. The key implication of this assumption is that A_0 can create a copy of its own operator and problem space hierarchy to model the opponent. (This creation of a copy by itself may not be straightforward if all of A_0's knowledge is essentially procedural.) This assumption essentially substitutes for another assumption in the plan recognition literature: the agent that is recognizing a plan is assumed to have full knowledge of all of the plans that the planning agent can execute [12]. If A_0 has such additional knowledge about how its opponent's plans or operators, and how those differ differ from its own, then A_0 need the ability to interleave those with its own copy of operators while tracking opponent's behaviors. If A_0 does not have this additional knowledge, then A_0 will need to model its opponent with incomplete information, or to learn that information from observation of the opponent's actions or by some other means.

3.4. A Prototype WCPS-based Agent

An important test of the WCPS model is its actual application in a dynamic, multi-agent environment. The task of developing an automated pilot for the air-combat simulation domain is tailor-made for this test. The development of automated pilots in this domain is currently based on a system called TacAir-Soar [11, 17], which as mentioned earlier, is developed using the Soar integrated problem-solving and learning architecture. TacAir-Soar is a "non-trivial" system that includes about 800 rules.[4] Its original self-centered problem space design worked against an initial inactive opponent. However, it very quickly failed against an active opponent — there was a need for tracking events related to actions of the other agents.

To survive in this real-time environment, the system was forced to employ world-centered problem spaces. However, these world-centered problem-spaces are created based on an incomplete and ad-hoc mechanism, that suffers from three problems. First, event tracking is not robust, meaning the automated pilot agent can and does generate unuseful or misleading interpretations for key opponent actions, such as the opponent's turn in Figure 1-c. This lack of robustness also implies that the automated pilot is unable to deal with sensor limitations effectively. Thus, sometimes if radar contact is momentarily lost, the agent may not track the opponent's actions. A second problem with the existing world-centered problem spaces is that event tracking does not generate expectations. A third problem is that the agent's real-time response can suffer due to sequential operator execution.

We have implemented a variant of TacAir-Soar that is fully based on WCPS. To create this variant, we started with the operators and problem spaces that are used by a TacAir-Soar-based automated pilot in generating its flexible actions and behaviors. We then generated a copy of these operators and problem spaces to model the automated pilot's opponent within a single WCPS. This copy was hand generated (since most of TacAir-Soar's knowledge is procedural, automatic generation of such a copy is an interesting research question that is left for future work). In generating this copy, some of TacAir-Soar's operators and problem spaces were abstracted away — these opponent actions were not modeled in detail. The result is an implementation that is able to track events while generating expectations. It is also promising in terms of being more robust in tracking events. The implementation tracks opponent's action and behavior as described provided in Section 3.2. Simultaneously, as discussed in Section 3.3, it avoids the communication overheads and duplication of information. The implementation currently only works in single opponent situation. Work on extending the implementation to multiple opponent situations is currently in progress.

[4]Since the completion of the experiment described in this section, the size of the TacAir-Soar system has grown to about 1500 rules.

4. Summary

This paper makes two contributions. First, it presents a detailed analysis of event tracking in the "real-world", dynamic, multi-agent environment of air-combat simulation. This analysis reveals interesting issues that represent a novel intersection of the areas of plan recognition and DAI. Tools and techniques that have emerged from single-agent environments are inadequate to address these issues. The second contribution of the paper is the idea of world-centered problem spaces (WCPS), for use in general multi-agent situations. WCPS is independent of problem spaces as such — the key idea is that an agent treats the generation of its own behavior and tracking of others uniformly. WCPS was used in (re)implementing automated pilots for air-combat simulation.

The paper also outlined several unresolved issues in WCPS. Among them, resolving ambiguity in opponent's actions, generating approximations, learning about the opponent from observation, and so on. We hope that addressing these issues will help in allowing WCPS to perform event tracking in a more robust fashion.

References

1. Anderson, J. R., Boyle, C. F., Corbett, A. T., and Lewis, M. W. "Cognitive modeling and intelligent tutoring". *Artificial Intelligence 42* (1990), 7-49.

2. Azarewicz, J., Fala, G., Fink, R., and Heithecker, C. Plan recognition for airborne tactical decision making. National Conference on Artificial Intelligence, 1986, pp. 805-811.

3. Carberry, S. Incorporating default inferences into Plan Recognition. Proceedings of National Conference on Artificial Intelligence, 1990, pp. 471-478.

4. Carberry, S. *Plan Recognition in Natural Language Dialogue.* MIT Press, Cambridge, MA, 1990.

5. Decker, K., and Lesser, V. Quantitative modeling of complex computational task environments. Proceedings of the National Conference on Artificial Intelligenence, 1993.

6. Dousson, C., Gaborit, P., and Ghallab, M. Situation Recognition: Representation and Algorithms. International Joint Conference on Artificial Intelligence, 1993, pp. 166-172.

7. Durfee, E. H., and Lesser, V. R. Using Partial Global Plans to Coordinate Distributed Problem Solvers. In Bond, A. H., and Gasser, L., Ed., *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann Publishers, Palo Alto, CA, 1988.

8. Forbus, K. "Qualitative Process Theory". *Artificial Intelligence 24* (1984), 85-168.

9. Grosz, B. J., and Sidner, C. L. Plans for Discourse. In *Intentions in Communication*, MIT Press, Cambridge, MA, 1990, pp. 417-445.

10. Hill, R., and Johnson, W. L. Impasse-driven tutoring for reactive skill acquisition. Proceedings of the Conference on Intelligent Computer-aided Training and Virtual Environment Technology, 1993.

11. Jones, R. M., Tambe, M., Laird, J. E., and Rosenbloom, P. Intelligent automated agents for flight training simulators. Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation, March, 1993.

12. Kautz, A., and Allen J. F. Generalized plan recognition. National Conference on Artificial Intelligence, 1986, pp. 32-37.

13. Laird, J.E. and Rosenbloom, P.S. Integrating execution, planning, and learning in Soar for external environments. Proceedings of the National Conference on Artificial Intelligence, July, 1990.

14. Laird, J. E., Newell, A. and Rosenbloom, P. S. "Soar: An architecture for general intelligence". *Artificial Intelligence 33*, 1 (1987), 1-64.

15. Rosenbloom, P. S., Laird, J. E., Newell, A., and McCarl, R. "A preliminary analysis of the Soar architecture as a basis for general intelligence". *Artificial Intelligence 47*, 1-3 (1991), 289-325.

16. Song, F. and Cohen, R. Temporal reasoning during plan recognition. National Conference on Artificial Intelligence, 1991, pp. 247-252.

17. Tambe, M., Jones, R., Laird, J. E., Rosenbloom, P. S., and Schwamb, K. Building Believable Agents for Simulation Environments. Proceedings of the AAAI Spring Symposium on Believable Agents, 1994. (to appear).

18. Van Beek, P., and Cohen, R. Resolving Plan Ambiguity for Cooperative Response Generation. Proceedings of International Joint Conference on Artificial Intelligence, 1993, pp. 938-944.

19. Ward, B. *ET-Soar: Toward an ITS for Theory-Based Representations.* Ph.D. Th., School of Computer Science, Carnegie Mellon University, May 1991.

Milind Tambe is a computer scientist at the Information Sciences Institute, University of Southern California (USC) and a research assistant professor with the computer science department at USC. He completed his undergraduate education in computer science from the Birla Institute of Technology and Science, Pilani, India in 1986. He received his Ph.D. in 1991 from the School of Computer Science at Carnegie Mellon University, where he continued as a research associate until 1993. His interests are in the areas of integrated AI systems, and efficiency and scalability of AI programs, especially rule-based systems.

Paul S. Rosenbloom is an associate professor of computer science at the University of Southern California and the acting deputy director of the Intelligent Systems Division at the Information Sciences Institute. He received his B.S. degree in mathematical sciences from Stanford University in 1976 and his M.S. and Ph.D. degrees in computer science from Carnegie-Mellon University in 1978 and 1983, respectively. His research centers on integrated intelligent systems (in particular, Soar), but also covers other areas such as machine learning, production systems, planning, and cognitive modeling. He is a Councillor of the AAAI and a past Chair of ACM SIGART.

A Very Low Cost System for Direct Human Control of Simulated Vehicles

Michael van Lent and Robert Wray
Artificial Intelligence Laboratory
University of Michigan
1101 Beal Avenue
Ann Arbor, MI 28109-2110
vanlent@eecs.umich.edu and wrayre@eecs.umich.edu

Abstract

Testing and knowledge acquisition have been two of the most tedious and time consuming tasks in the development of IFOR agents in the TacAir-Soar (TAS) project. This paper presents some suggestions for a human control tool, similar to a simple flight simulator, that can be helpful in these two areas. Furthermore, we discuss some of the design considerations and implementation issues that are faced in developing such a tool. Such a tool, called the Human Instrument Panel (HIP) has been developed for use in the TacAir-Soar project. Some key features of HIP are how cheaply it has been developed, how quickly is was incorporated into the TacAir-Soar project, and how easily it can be adapted to similar domains.

Introduction

The Human Instrument Panel (HIP) is a tool designed to make the testing of TacAir-Soar (TAS) agents in the ModSAF simulator easier and less tedious. Additionally it has been useful to a lesser extent as an aid in the lengthy process of knowledge acquisition. This paper presents suggestions as to how this type of tool can be used as an aid in testing and knowledge acquisition and discusses some of the design considerations that the creator of such a tool might face. Since the Human Instrument Panel is meant to be a time-saving tool, one major design consideration is that the time and effort saved by HIP outweigh the time and effort spent on development and maintenance.

The TacAir-Soar project[1] at the University of Michigan, ISI, and Carnegie Mellon University has combined Soar, a state of the art artificial intelligence architecture, and ModSAF, a sophisticated battlefield simulator, to create realistic, human-like computer agents in a beyond visual range air-to-air combat domain. The development of these agents can be viewed as a repeated cycle through three phases[2]:

1. Knowledge Acquisition
2. Implementation
3. Testing

Usually the majority of the development time in the TacAir-Soar project is spent in the knowledge acquisition and testing phases. It is for this reason that HIP has been developed specifically to assist in these two tasks.

The main difficulties in knowledge acquisition are the vast amount of information that must be acquired and the formulation of questions to extract the most important information. One effective form of knowledge acquisition, which sidesteps the questions formulation difficulty, is to observe expert pilots as they fly missions on simulators. This allows the questioner to identify issues that might never come up in a question and answer session. Unfortunately, the cost of running the simulators and getting the pilots and researchers to the simulator site does not allow for the amount of free play that would be required to cover the breadth of necessary knowledge.

Like knowledge acquisition, one of the difficulties of the testing phase is getting the experts, researchers, and machines together so that the experts can evaluate the TacAir-Soar agents. Another major difficulty is the lack of a flexible, realistic opponent against which to test the TAS agents. ModSAF controlled agents are not sufficiently intelligent to provide realistic challenges to the TAS agents and there is no support in ModSAF for direct human control of agents (i.e. no flight simulator capability). Until recently, testing a TAS agent in a specific scenario required creating a separate set of TAS agents as opponents with hard-coded missions, and then carefully designing the initial situation so that the desired scenario would occur. This was a time consuming process and testing TAS agents only against other TAS agents left the possibility of undetected errors.

One obvious solution to these problems is to create a tool which acts as a simple flight simulator interface to ModSAF agents. This would provide the ability to observe free play sessions without expensive simulators (TacAir-Soar has been tested against BATTs[1] simulators at the WISSARD lab at the Oceana Naval base). Additionally, the interface could also serve as a realistic opponent for TAS agents, pointing out errors that TAS vs. TAS testing might miss. This is exactly the role that the Human Instrument Panel (HIP) is designed to fill. HIP allows the user to attach a simple instrument panel to a ModSAF agent and issue flight commands to that agent's plane.

One goal of this paper is to describe the wide range of possible uses for a human control tool such as HIP in the creation of intelligent forces (IFORs). Hopefully the ways in which we have found HIP to be useful will suggest techniques that will make testing and knowledge acquisition for other IFORs easier. Some of the design considerations involved in creating human control tools will be discussed along with the pros and cons of the choices made while developing HIP.

The next section of this paper will describe the potential uses for human control tools, such as HIP, both as testing tools and aids to knowledge acquisition. Section 3 will point out some of the important design considerations and discuss the advantages of various approaches while section 4 provides a quick description of the various forms of HIP that we have developed (F-14D, MiG-29, E-2C) including screen snapshots of HIP in action.

Functionality of a Human Control Tool

When we set out to develop the Human Instrument Panel we were motivated by the need for a flexible and realistic opponent for TAS agents. As the project progressed we came up with many more potential uses which required only minor additions to the original specifications. The functionality of human control tools can be divided into two categories: testing aids and knowledge acquisition aids. Since HIP was designed mainly as a tool for testing we will focus primarily on its applications to this phase. There are three major components of IFOR development in the Tactical Air domain that HIP facilitates: testing in a variety of situations, monitoring an agent's local (instrument-level) behavior during testing, and scenario setup.

Testing Intelligent Agents HIP enables three different types of testing that are beneficial to the IFOR designer. First, one would like to test the performance of an agent against a human pilot. Such tests may involve determining the response of the agent to specific tactical situation (e.g., bogey approaches TAS from the right, rear quarter and performs a set series of maneuvers). With HIP, the researcher can take control of a plane and use HIP to approach the TAS agent from the specified quarter and perform the required maneuvers. The TAS agent's responses to these actions can be recorded for later evaluation. This is an example of scripted testing. The nature of the HIP interface also encourages free-play testing. One can simply create a TacAir-Soar agent, create a HIP agent, and then fly, head-to-head. While on the surface this may seem more like play than research, this can lead to the observation of behaviors not explicitly seen during scripted testing.

A second type of testing that can be accomplished with HIP is an agent's ability to coordinate its actions with other agents. This is done by allowing a human and a TacAir-Soar agent to fly together. Communication from HIP to the agent is accomplished using a series of pull-down menus that correspond to the types and formats of radio messages the TacAir-Soar agents can send to one another. The advantage of testing this coordination with a human pilot (as opposed to two IFORs interacting) is that specific tests can be scripted that would be difficult to carry out with two IFORs. For example, consider the question of testing an agent's behavior when its wingman is lost. Using an IFOR for this test would require implementing an agent that would purposely lose its lead. Similar tests in the BATTs are impossible since there is no communication interface between that simulator and Soar. However, using HIP in this test requires only that the pilot acting as the wingman fly away after establishing a communication link with the lead.

A third type of testing facilitated by HIP is the ability to monitor the response of IFOR agents in situations with varying world knowledge. For example, an agent's behavior toward a single contact should probably be modified if the agent is informed of multiple, hostile contacts beyond radar visibility. This ability to control the agent's world knowledge is achieved in HIP by introducing a new plane type, the E-2C[2]. The HIP E-2C can direct BRASH (Bearing-Range-Altitude-Speed-Heading) contact information to TacAir-Soar agents as well as to other HIP agents (and, conceivably, to BATTs pilots as well). The HIP E-2C also is complementing the design of an E-2C TacAir-Soar agent. Prior to the implementa-

[1]The Basic Air Tactics Trainer is a medium-fidelity aircraft simulator.

[2]The E-2C is a prop-driven, non-combat plane with a large AWACS-style radar.

tion of the HIP E-2C, there was no way to inform TacAir-Soar agents about contacts out of their radar range.

Monitoring An Intelligent Agent's Behavior In ModSAF each unit is displayed as an icon with associated heading, speed, and altitude values. This is sufficient for observing a TacAir-Soar agent's high level behavior but it does not provide much information about how realistically the agent is flying nor any important information such as radar modes and weapon selection. Another possible use of a human control tool is to "peek over TacAir-Soar's shoulder" as the TAS agent flies by displaying that plane's instrument readouts while leaving the agent in control. To support this an option was added to HIP to attach the instrument panel to a plane but suppress the transmission of any flight commands from HIP. This has provided some useful feedback as to how certain flight dynamics are handled in ModSAF. Additionally expert pilots may find it easier to evaluate TacAir-Soar agents from the familiar (somewhat realistic) cockpit perspective.

Building Scenarios The ability to selectively suppress or allow the transmission of flight commands from HIP could also very useful when setting up scenarios to test an agent's response to specific situations. It is often difficult to set the initial position of each unit involved in the scenario so that a desired encounter occurs. With the human control tool it could be possible to take control of each agent and fly it into exactly the position required and then return control to TAS. Once the user has taken control away from TAS, HIP does not currently allow control to be returned to TAS. This is due to the difficulty of keeping the TAS agent's internal state consistent with the external world. Once this problem is overcome, it is possible that TAS could learn behaviors by "observing" while a human flies the agent's plane and executes the desired actions.

Design Considerations

There are obviously many ways that a human control tool can be implemented, each with various advantages and disadvantages. In this section we will discuss a few of these implementation decisions as well as some important high-level design considerations.

High-Level Considerations Although the anticipated uses of HIP drove its design, several other high-level factors had to be considered as well. Four are discussed here: level of detail in the simulation, non-invasive interface with the simulator, shallow learning curve for new users, and a simple implementation.

In considering the level of cockpit detail for the HIP interface, one key decision was made which affected the subsequent development of the tool. The TacAir-Soar agents use a *cockpit abstraction*[1] to interface with ModSAF. The agents send commands to ModSAF indicating flight parameters such as desired altitude and speed; ModSAF includes functions to convert these high-level commands into low-level flight surface, sensor and weapon controls. Since one of the goals of the HIP project was to produce a realistic tool as quickly as possible, the TAS/ModSAF interface was adopted for HIP as well. This resulted in a less-realistic interface than many flight simulators – there is no joystick and commands are entered for the desired heading, altitude and speed while ModSAF determines the appropriate flight response. This decision does represent a compromise in simulator realism. However, since the domain of interest is tactical rather than low-level flight, this choice allowed much faster development while not compromising HIP's anticipated uses.

Because HIP was to be used in a simulated environment with an unknown (and possibly large) number of other agents, HIP could not adversely affect the normal operation of ModSAF; the added functionality had to be non-invasive. Similarly, HIP also had to be transparent to ModSAF so that ModSAF would work normally if no HIP agent were needed. The implementation decisions outlined below allowed HIP to achieve these requirements. Finally, since HIP is usually run from the same workstation as the simulator, its display had to minimize interference with the ModSAF display. This drove the decision to make the HIP F-14 and MiG-29 windows as small as possible.

Another factor in the design of HIP was that it needed to be simple to use. Building an interface that required pilot skill would have defeated many of the functionalities described above. This design constraint was met by using a graphical interface built to resemble a highly schematic cockpit. Flight controls are modified using a mouse with "click-and-drag" widgets. This allows a novice user to receive instruction on HIP and be "up and flying" in less than five minutes. Additionally, demonstrations and reviews can now include sessions in which on-lookers can participate in an engagement — with or against – an IFOR agent.

Finally, the time required to create HIP had to be considered, taking into account all the design goals and constraints mentioned heretofore. HIP is intended as a tool to aid IFOR research and not an end unto itself. The system had to be developed quickly, with a minimum impact on project personnel. Additionally, the completed system had to support the addition of features and enhancements without significant effort. The basic implementation strategy derives from this constraint.

Human Control Tool/Simulator Interface One of the first issues faced in the development of HIP was how it would communicate with ModSAF. One possibility was to make HIP a part of the ModSAF executable. The advantage of this approach was that communication could be accomplished via parameters to function calls and would allow for very fast transmission and as high a bandwidth as necessary. The disadvantages were that making HIP a part of the ModSAF application would make the executable larger and slower. As mentioned above one of our high level design goals was to limit any adverse effects HIP might have on the ModSAF system. Because of this, HIP was not incorporated into the ModSAF application.

The approach we chose was to make HIP a separate executable which communicated with ModSAF through UNIX sockets. A variety of interprocess communication packages would have been appropriate but sockets were chosen because working code was available. The main advantage of the separate process approach is that HIP can be run on a separate machine and therefore has very little effect on the speed of ModSAF. Also the addition of the HIP communication interface added only 50 kilobytes to the ModSAF executable while the entire HIP package would have added well over a megabyte. The transfer rate through the sockets seems well able to keep up with ModSAF; however, if this becomes a problem more efficient interprocess communication techniques are available.

Moded vs. Unmoded User Interface In addition to these requirements, the control structure of the final interface needed to be unmoded. A moded system is simply one in which certain actions can be made only when initiated by a previous series of actions. Conversely, an unmoded design allows most functionality to be accessed independent of other actions. This capability was important for HIP since most actions occur as a response to the current situation. For example, a pilot may decide to turn in response to a number of different situations: firing a missile, evading an approaching missile, receiving an order from the lead or taking advantage of the tactical situation. Thus, it is important that a HIP pilot be able to turn at any time. Such behavior is supported by HIP's unmoded design. Specifically, all controls in the HIP interface can be accessed at any time. Controls that require a series of steps (e.g., loading and then firing a weapon) must be ordered by the HIP pilot. Therefore, pressing the fire button has no effect when a missile is not loaded. Such an umoded design is consistent with an actual cockpit and adds to HIP's ease-of-use.

Quick and Cheap Development The HIP interface required a sophistication in computer graphics that would have required either expensive consultation or time for the designers to learn such sophistication. However, there are many software packages available that allow the design of user interfaces at the "widget" level rather than the pixel level of most computer graphics programming. A widget is simply a pre-defined graphics component with a specific functionality such as a menu or slide-bar. Examples of such widget design packages include X-Designer[3] (Imperial Software Technologies), Builder Xcessory[4] (Integrated Computer Solutions), and the Simple User Interface Toolkit[5] or SUIT (University of Virginia). SUIT was chosen for this project because it was available to the university free-of-charge and it included the following needed features: a reasonable assortment of widgets, the ability to design widgets with specific functionality, and good documentation backed by a large user group.

The ability to create user-defined widgets was particularly important. For example, the first implementation of the radar display contained only textual information and proved difficult to use. This information was encoded in the subsequent design of the graphical radar display. The color of a particular radar contact is used to represent the classification of an agent as friendly, enemy or unknown. The shape of the contact determines if the contact is held via radar, visual or both. A vector from the contact gives relative heading and speed information. This display has proven simple to use, conveying a great deal of information via this customization of the widget. Additionally, the capability to select targets by clicking on them in the radar display was added. This removed the necessity of identifying agents by call sign or vehicle-id when targeting. Other user-defined widgets include the heading display and the HUD. These widgets increase both the functionality and usability of the interface but, because SUIT supports the design of such widgets, does not require programming at the pixel level for such increased capability.

The first implementation of HIP (for the F-14) was done by two graduate students, working part-time over the course of a semester (approximately three effort-months). This included the full functionality of the agent described in Section 4 as well as researching the capabilities and payloads of the actual aircraft[6,7]. Modifying HIP for a similar plane (the MiG-29) took about forty-five minutes of actual coding and about a day to test after researching the appropriate flight and weapons parameters. Finally, incorporating a completely different plane-type, with both added features and a different functionality (the E-2C), took only 10 hours of total effort, again after the appropriate research had been completed. These efforts showed that our design goals had been met: a tool had been developed quickly that could be used in number of ways and that did not interfere with the simulator. Additionally, the design criteria allowed the interface to be modified very quickly for application to slightly different agents in the same domain.

Description of HIP

The F-14 version was the first HIP version constructed (see Figure 1). The display is divided into three sections. The left-most section includes the communications interface and widgets for selecting and deleting HIP agents (HIP may be used to control more than one F-14 at a time). Communications is accomplished via a series of windows that represent message templates. For example, when the "Current Position" message is selected, HIP automatically fills in that information in the template. Messages not corresponding to the template messages can also be entered. In the center of the display are the flight controls as well as buttons for releasing control of an agent from ModSAF (toggling the transmission of commands to ModSAF) and quitting ModSAF. There is also a simple Heads Up Display (HUD) in the center of the window. The flight controls for heading, altitude and speed include both the current value (given by the large, filled arrows) and the desired value (the small arrow in the heading display, the position of the sliders for altitude and speed). Finally, the right-most section of the HIP window consists of the radar display, radar controls, and weapons controls. As mentioned previously, targeting simply consists of clicking on contacts in the radar display.

Figure 2 is an example of the HIP E-2C display. The reduced-function flight controls have been placed to the right and the radar display enlarged and moved to the center. Weapons controls have been deleted and a new window created for displaying the BRASH of the current contact-of-interest (COI). This information is generated automatically when BRASH information is "radioed" to a TacAir-Soar agent. BRASH information can be generated with respect to either the position of the E-2C or another agent. The HIP E-2C, with both a different display and different functionality from the HIP F-14 agent, was created by utilizing the basic, generic structure that was purposely used in building the F-14 agent.

Flying in HIP simply requires entering the name of an appropriate agent from ModSAF in the *Select Plane* text box, setting the desired starting configuration, and then hitting the *Take Control* button. Once control has been established the user can then hit the button again to *Release Control*. As mentioned previously, the user's activity while flying is unmoded and any action can be taken immediately in response to the user's evaluation of the current scenario.

Although the flight controls are rudimentary, experimentation with HIP has shown that sophisticated maneuvers can be accomplished. However, in addition to these maneuvers, inexperienced pilots also often make mistakes. One of the most obvious is turn too hard, too often, resulting in stalls. Stalls may be recovered by diving hard until speed increases sufficiently for re-engaging the engines. What is interesting about these maneuvers is that by just using HIP and getting a "feel" for flying, TAS designers have become more comfortable with the problem domain and have gained insights into many of its features and limitations. This has proved an unanticipated benefit of HIP but one that is proving useful, especially as the agents are modeled at increasing levels of detail.

Conclusion

While the examples in this paper have concentrated on the air to air combat domain, human control tools can be easily transferred to other domains where a similar interaction is desired between human and computer generated forces. The near-immediate extensions to HIP for the MiG-29 and E-2C demonstrate the extensibility of the basic tool. In the future we hope to extend HIP by creating versions for close air support units, ground forces and other vehicles supported by ModSAF by utilizing the underlying ModSAF functions for low-level agent behavior. Thus, having invested in the implementation of the basic tool, application to different domains is considerably simplified.

This paper has discussed some of the decisions appropriate for developing a simple interface for human interaction with intelligent forces. These decisions were constrained by the following questions:

- What functions should be supported by the tool?
- How much effort can be invested in tool development?
- What tools are available to make development

Figure 1: The HIP F-14 Instrument Panel.

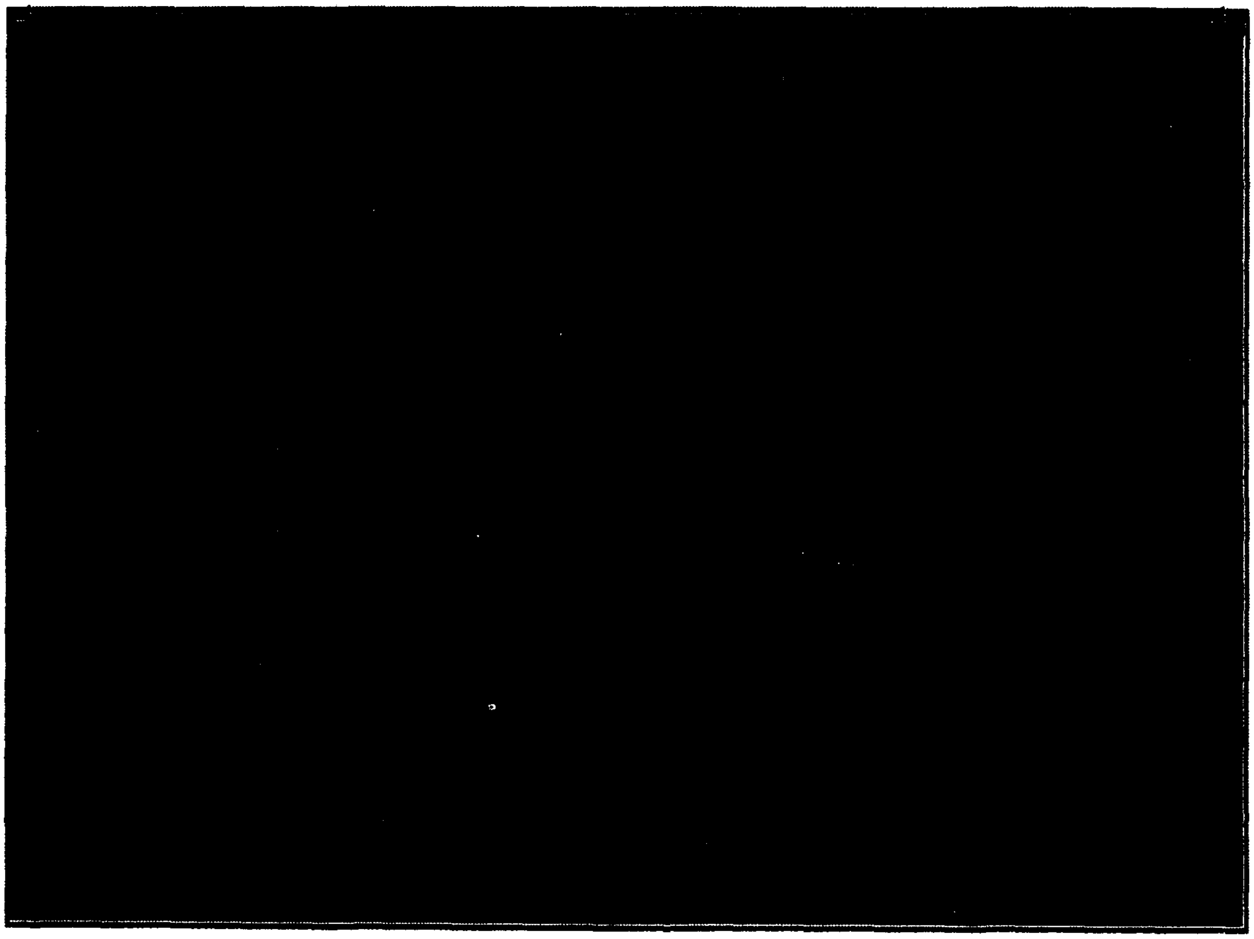


Figure 2: The E-2C Radar Controller Window and Flight Controls.

quick, inexpensive and robust?

In attempting to explore the trade-offs and explain the rationale behind our answers we hope we have provided a motivation and framework for the development of similar tools. We have presented the Human Instrument Panel as one example of such a tool and described a wide variety of ways in which such a human control tool can aid in the testing and knowledge acquisition necessary for any IFOR project.

Acknowledgements

This research was supported under contract to the University of Michigan, as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). Critical support has been provided by John Laird, Paul Rosenbloom, Randy Jones, Frank Koss, Karl Schwamb and the SUIT Users Group.

References

[1] Rosenbloom, P., Johnson, L., Jones, R., Koss, F., Laird, J., Lehman, J., Rubinoff, R., Schwamb, K., Tambe, M. ***Intelligent automated agents for tactical air simulation: a progress report.*** Manuscript submitted for publication.

[2] Jones, R., Tambe, T., Laird, J., Rosenbloom, P. Intelligent automated agents for flight training simulators. In ***Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation***. Univ. of Central FL, IST-TR-93-07, 1993.

[3] Conway, M., Pausch, R., Passarella, K. ***The SUIT Version 2.3 Reference Manual.*** University of Virginia, 1992.

[4] ***X-designer user's manual, release 3.*** Imperial Software Technology, 1993.

[5] ***Builder Xcessory user's manual.*** Integrated Computer Solutions, Incorporated. Cambridge, MA, 1992.

[6] Brinkman, D., ed. ***Jane's Avionics, 12th Edition.*** Jane's Information Group, Inc. 1993.

[7] Stevenson, J. ***Grumman F-14 Tomcat.*** McGraw-Hill Books, Blue Ridge Summit, PA. 1975.

Biographies

Michael van Lent is currently a doctoral student in the Artificial Intelligence Laboratory at the University of Michigan. He received his B.A. with honors in computer science from Williams College in 1991 and a Master of Science in Computer Science from the University of Tennessee, Knoxville in 1993. Mr. van Lent also worked for the Naval Center for Applied Research in Artificial Intelligence during the summers of 1992 and 1993.

Robert Wray is currently a doctoral student at the University of Michigan. Mr. Wray received a Bachelor of Science in Electrical Engineering from Memphis State University in 1988 and a Master of Science in Electrical Engineering from the University of Massachusetts, Dartmouth in 1993. He also worked for the Naval Undersea Warfare Center from 1989 to 1993, focusing primarily on the automatic generation of presets for submarine-launched weapons.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890